

Jennifer Welch (Ed.)

LNCS 2180

# Distributed Computing

15th International Conference  
Lisbon, Portugal, October 2-6, 2000  
Proceedings

# Lecture Notes in Computer Science

Edited by G. Goos, J. Hartmanis, and J. van Leeuwen

2180

**Springer**

*Berlin*

*Heidelberg*

*New York*

*Barcelona*

*Hong Kong*

*London*

*Milan*

*Paris*

*Tokyo*

Jennifer Welch (Ed.)

# Distributed Computing

15th International Conference, DISC 2001  
Lisbon, Portugal, October 3-5, 2001  
Proceedings



Springer

## Series Editors

Gerhard Goos, Karlsruhe University, Germany  
Juris Hartmanis, Cornell University, NY, USA  
Jan van Leeuwen, Utrecht University, The Netherlands

## Volume Editor

Jennifer Welch  
Texas A&M University, Department of Computer Science  
College Station, TX 77843-3112, USA  
E-mail: [welch@cs.tamu.edu](mailto:welch@cs.tamu.edu)

## Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Distributed computing : 15th international conference ; proceedings / DISC  
2001, Lisbon, Portugal, October 3 - 5, 2001. Jennifer Welch (ed.). - Berlin ;  
Heidelberg ; New York ; Barcelona ; Hong Kong ; London ; Milan ; Paris ;  
Tokyo : Springer, 2001  
(Lecture notes in computer science ; Vol. 2180)  
ISBN 3-540-42605-1

CR Subject Classification (1998): C.2.4, C.2.2, F.2.2, D.1.3, F.1, D.4.4-5

ISSN 0302-9743

ISBN 3-540-42605-1 Springer-Verlag Berlin Heidelberg New York

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of translation, reprinting, re-use of illustrations, recitation, broadcasting, reproduction on microfilms or in any other way, and storage in data banks. Duplication of this publication or parts thereof is permitted only under the provisions of the German Copyright Law of September 9, 1965, in its current version, and permission for use must always be obtained from Springer-Verlag. Violations are liable for prosecution under the German Copyright Law.

Springer-Verlag Berlin Heidelberg New York  
a member of BertelsmannSpringer Science+Business Media GmbH

<http://www.springer.de>

© Springer-Verlag Berlin Heidelberg 2001  
Printed in Germany

Typesetting: Camera-ready by author, data conversion by PTP-Berlin, Stefan Sossna  
Printed on acid-free paper      SPIN: 10840494      06/3142      5 4 3 2 1 0

# Preface

DISC, the International Symposium on DIStributed Computing, is an annual forum for research presentations on all facets of distributed computing. DISC 2001 was held on Oct 3–5, 2001, in Lisbon, Portugal. This volume includes 23 contributed papers. It is expected that these papers will be submitted in more polished form to fully refereed scientific journals. The extended abstracts of this year’s invited lectures, by Gerard LeLann and David Peleg, will appear in next year’s proceedings.

We received 70 regular submissions. These submissions were read and evaluated by the program committee, with the help of external reviewers when needed. Overall, the quality of the submissions was excellent, and we were unable to accept many deserving papers.

This year’s *Best Student Paper* award goes to Yong-Jik Kim for the paper “A Time Complexity Bound for Adaptive Mutual Exclusion” by Yong-Jik Kim and James H. Anderson.

October 2001

Jennifer Welch

## Organizing Committee

Chair:	Luis Rodrigues (University of Lisbon)
Publicity:	Paulo Veríssimo (University of Lisbon)
Treasurer:	Filipe Araújo (University of Lisbon)
Web:	Alexandre Pinto (University of Lisbon)
Registration:	Hugo Miranda (University of Lisbon)

## Steering Committee

Faith Fich (U. of Toronto)	Michel Raynal (vice-chair) (IRISA)
Maurice Herlihy (Brown U.)	André Schiper (chair) (EPF Lausanne)
Prasad Jayanti (Dartmouth)	Jennifer Welch (Texas A&M U.)
Shay Kutten (Technion)	

## Program Committee

Marcos K. Aguilera (Compaq SRC)  
Mark Moir (Sun Microsystems Laboratories)  
Lorenzo Alvisi (U. Texas, Austin)  
Stephane Perennes (CNRS U. de Nice INRIA)  
Hagit Attiya (Technion)  
Benny Pinkas (STAR Lab, Intertrust Technologies)  
Shlomi Dolev (Ben-Gurion U.)  
Ulrich Schmid (Technical U., Vienna)  
Tamar Eilam (IBM T.J. Watson Research Center)  
Philippas Tsigas (Chalmers U.)  
Amr El-Abbadi (U. California, Santa Barbara)  
Jennifer Welch (chair) (Texas A&M U.)  
Panagiota Fatourou (Max-Planck Inst. Informatik)  
Pierre Wolper (U. Liege)  
John Mellor-Crummey (Rice U.)

## Outside Referees

Adnan Agbaria  
Fred Annexstein  
Ken Arnold  
Joshua Auerbach  
Zvi Avidor  
Sumeer Bhola  
Vita Bortnikov  
B. Charron-Bost  
Wei Chen  
Murat Demirbas  
J. Durand-Lose  
Shimon Even  
Alan Fekete  
Guillaume Fertin  
Faith Fich  
Arie Fouren  
Roy Friedman  
Eli Gafni

Juan Garay  
Cyril Gavoille  
German Goldszmidt  
Abhishek Gupta  
Indranil Gupta  
Vassos Hadzilacos  
Maurice Herlihy  
S. T. Huang  
Colette Johnen  
Michael Kalantar  
Idit Keidar  
Roger Khazan  
Ami Litman  
Keith Marzullo  
Marios Mavronicolas  
Giovanna Melideo  
Mikhail Nesterenko  
Ronit Nosenon

Rafail Ostrovsky  
M. Paramasivam  
Boaz Patt-Shamir  
Andrzej Pelc  
Eric Ruppert  
André Schiper  
Peter Sewell  
Nir Shavit  
Alex Shvartsman  
Cormac J. Sreenan  
Rob Strom  
Tami Tamir  
Mark Tuttle  
John Valois  
Roman Vitenberg  
Avishai Wool



# Table of Contents

A Time Complexity Bound for Adaptive Mutual Exclusion . . . . .	1
<i>Y.-J. Kim and J.H. Anderson</i>	
Quorum-Based Algorithms for Group Mutual Exclusion . . . . .	16
<i>Y.-J. Jourg</i>	
An Effective Characterization of Computability in Anonymous Networks . . . . .	33
<i>P. Boldi and S. Vigna</i>	
Competitive Hill-Climbing Strategies for Replica Placement in a Distributed File System . . . . .	48
<i>J.R. Douceur and R.P. Wattenhofer</i>	
Optimal Unconditional Information Diffusion . . . . .	63
<i>D. Malkhi, E. Pavlov, and Y. Sella</i>	
Computation Slicing: Techniques and Theory . . . . .	78
<i>N. Mittal and V.K. Garg</i>	
A Low-Latency Non-blocking Commit Service . . . . .	93
<i>R. Jiménez-Peris, M. Patiño-Martínez, G. Alonso, and S. Arévalo</i>	
Stable Leader Election . . . . .	108
<i>M.K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg</i>	
Adaptive Long-Lived $O(k^2)$ -Renaming with $O(k^2)$ Steps . . . . .	123
<i>M. Inoue, S. Umetani, T. Masuzawa, and H. Fujiwara</i>	
A New Synchronous Lower Bound for Set Agreement . . . . .	136
<i>M. Herlihy, S. Rajsbaum, and M. Tuttle</i>	
The Complexity of Synchronous Iterative Do-All with Crashes . . . . .	151
<i>C. Georgiou, A. Russell, and A.A. Shvartsman</i>	
Mobile Search for a Black Hole in an Anonymous Ring . . . . .	166
<i>S. Dobrev, P. Flocchini, G. Prencipe, and N. Santoro</i>	
Randomised Mutual Search for $k > 2$ Agents . . . . .	180
<i>J.-H. Hoepman</i>	
Self-Stabilizing Minimum Spanning Tree Construction on Message-Passing Networks . . . . .	194
<i>L. Higham and Z. Liang</i>	

Self Stabilizing Distributed Queuing . . . . .	209
<i>M. Herlihy and S. Tirthapura</i>	
A Space Optimal, Deterministic, Self-Stabilizing, Leader Election Algorithm for Unidirectional Rings . . . . .	224
<i>F.E. Fich and C. Johnen</i>	
Randomized Finite-State Distributed Algorithms as Markov Chains . . . . .	240
<i>M. Duflot, L. Fribourg, and C. Picaronny</i>	
The Average Hop Count Measure for Virtual Path Layouts . . . . .	255
<i>D. Peleg and U. Pincas</i>	
Efficient Routing in Networks with Long Range Contacts . . . . .	270
<i>L. Barrière, P. Fraigniaud, E. Kranakis, and D. Krizanc</i>	
An Efficient Communication Strategy for Ad-hoc Mobile Networks . . . . .	285
<i>I. Chatzigiannakis, S. Nikolettseas, and P. Spirakis</i>	
A Pragmatic Implementation of Non-blocking Linked-Lists . . . . .	300
<i>T.L. Harris</i>	
Stabilizing Replicated Search Trees . . . . .	315
<i>T. Herman and T. Masuzawa</i>	
Adding Networks . . . . .	330
<i>P. Fatourou and M. Herlihy</i>	
<b>Author Index . . . . .</b>	<b>343</b>

# A Time Complexity Bound for Adaptive Mutual Exclusion<sup>\*</sup>

(Extended Abstract)

Yong-Jik Kim and James H. Anderson

Department of Computer Science  
University of North Carolina at Chapel Hill

**Abstract.** We consider the time complexity of adaptive mutual exclusion algorithms, where “time” is measured by counting the number of remote memory references required per critical-section access. We establish a lower bound that precludes a deterministic algorithm with  $O(\log k)$  time complexity (in fact, any deterministic  $o(k)$  algorithm), where  $k$  is “point contention.” In contrast, we show that expected  $O(\log k)$  time *is* possible using randomization.

## 1 Introduction

In this paper, we consider the time complexity of adaptive mutual exclusion algorithms. A mutual exclusion algorithm is *adaptive* if its time complexity is a function of the number of contending processes [3,6,8,10,11]. Under the time complexity measure considered in this paper, only remote memory references that cause a traversal of the global processor-to-memory interconnect are counted. Specifically, we count the number of such references generated by one process  $p$  in a computation that starts when  $p$  becomes active (leaves its noncritical section) and ends when  $p$  once again becomes inactive (returns to its noncritical section). Unless stated otherwise, we let  $k$  denote the “point contention” over such a computation (the *point contention* over a computation  $H$  is the maximum number of processes that are active at the same state in  $H$  [1]). Throughout this paper, we let  $N$  denote the number of processes in the system.

In recent work, we presented an adaptive mutual exclusion algorithm — henceforth called ALGORITHM AK — with  $O(\min(k, \log N))$  time complexity [3]. ALGORITHM AK requires only read/write atomicity and is the only such algorithm known to us that is adaptive under the remote-memory-references time complexity measure. In other recent work, we established a worst-case time bound of  $\Omega(\log N / \log \log N)$  for mutual exclusion algorithms (adaptive or not) based on reads, writes, or comparison primitives such as test-and-set and compare-and-swap [4]. (A *comparison primitive* conditionally updates a shared variable after first testing that its value meets some condition.) This result shows

---

<sup>\*</sup> Work supported by NSF grants CCR 9732916, CCR 9972211, CCR 9988327, and ITR 0082866.

that the  $O(\log N)$  worst-case time complexity of ALGORITHM AK is close to optimal. In fact, we believe it *is* optimal: we conjecture that  $\Omega(\log N)$  is a tight lower bound for this class of algorithms.

If  $\Omega(\log N)$  is a tight lower bound, then presumably a lower bound of  $\Omega(\log k)$  would follow as well. This suggests two interesting possibilities: in all likelihood, either  $\Omega(\min(k, \log N))$  is a *tight* lower bound (*i.e.*, ALGORITHM AK is optimal), or it is possible to design an adaptive algorithm with  $O(\log k)$  time complexity (*i.e.*,  $\Omega(\log k)$  is tight). Indeed, the problem of designing an  $O(\log k)$  algorithm using only reads and writes has been mentioned in two recent papers [3,6].

In this paper, we show that an  $O(\log k)$  algorithm in fact does not exist. In particular, we prove the following: *For any  $k$ , there exists some  $N$  such that, for any  $N$ -process mutual exclusion algorithm based on reads, writes, or comparison primitives, a computation exists involving  $\Theta(k)$  processes in which some process performs  $\Omega(k)$  remote memory references to enter and exit its critical section.*

Although this result precludes a deterministic  $O(\log k)$  algorithm (in fact, any deterministic  $o(k)$  algorithm), we show that a randomized algorithm does exist with *expected*  $O(\log k)$  time complexity. This algorithm is obtained through a simple modification to ALGORITHM AK.

The rest of the paper is organized as follows. In Sec. 2, our system model is defined. Our lower bound proof is presented in Secs. 3-4. The randomized algorithm mentioned above is sketched in Sec. 5. We conclude in Sec. 6.

## 2 Definitions

Our model of a shared-memory system is based on that used in [4,5].

*Shared-memory systems.* A shared-memory system  $\mathcal{S} = (C, P, V)$  consists of a set of computations  $C$ , a set of processes  $P$ , and a set of variables  $V$ . A *computation* is a finite sequence of events.

An *event*  $e$  is denoted  $[R, W, p]$ , where  $p \in P$ . The sets  $R$  and  $W$  consist of pairs  $(v, \alpha)$ , where  $v \in V$ . This notation represents an event of process  $p$  that reads the value  $\alpha$  from variable  $v$  for each element  $(v, \alpha) \in R$ , and writes the value  $\alpha$  to variable  $v$  for each element  $(v, \alpha) \in W$ . Each variable in  $R$  (or  $W$ ) is assumed to be distinct. We define  $Rvar(e)$ , the set of variables read by  $e$ , to be  $\{v \mid (v, \alpha) \in R\}$ , and  $Wvar(e)$ , the set of variables written by  $e$ , to be  $\{v \mid (v, \alpha) \in W\}$ . We also define  $var(e)$ , the set of all variables accessed by  $e$ , to be  $Rvar(e) \cup Wvar(e)$ . We say that this event *accesses* each variable in  $var(e)$ , and that process  $p$  is the *owner* of  $e$ , denoted  $owner(e) = p$ . For brevity, we sometimes use  $e_p$  to denote an event owned by process  $p$ .

Each variable is *local* to at most one process and is *remote* to all other processes. (Note that we allow variables that are remote to *all* processes.) An *initial value* is associated with each variable. An event is *local* if it does not access any remote variable, and is *remote* otherwise.

We use  $\langle e, \dots \rangle$  to denote a computation that begins with the event  $e$ , and  $\langle \rangle$  to denote the empty computation. We use  $H \circ G$  to denote the computation

obtained by concatenating computations  $H$  and  $G$ . The value of variable  $v$  at the end of computation  $H$ , denoted  $value(v, H)$ , is the last value written to  $v$  in  $H$  (or the initial value of  $v$  if  $v$  is not written in  $H$ ). The last event to write to  $v$  in  $H$  is denoted  $writer\_event(v, H)$ , and its owner is denoted  $writer(v, H)$ . (Although our definition of an event allows multiple instances of the same event, we assume that such instances are distinguishable from each other.) If  $v$  is not written by any event in  $H$ , then we let  $writer(v, H) = \perp$  and  $writer\_event(v, H) = \perp$ .

For a computation  $H$  and a set of processes  $Y$ ,  $H|Y$  denotes the subcomputation of  $H$  that contains all events in  $H$  of processes in  $Y$ . Computations  $H$  and  $G$  are *equivalent* with respect to  $Y$  iff  $H|Y = G|Y$ . A computation  $H$  is a  $Y$ -*computation* iff  $H = H|Y$ . For simplicity, we abbreviate the preceding definitions when applied to a singleton set of processes. For example, if  $Y = \{p\}$ , then we use  $H|p$  to mean  $H|\{p\}$  and  $p$ -computation to mean  $\{p\}$ -computation.

The following properties apply to any shared-memory system.

- (P1) If  $H \in C$  and  $G$  is a prefix of  $H$ , then  $G \in C$ .
- (P2) If  $H \circ \langle e_p \rangle \in C$ ,  $G \in C$ ,  $G|p = H|p$ , and if  $value(v, G) = value(v, H)$  holds for all  $v \in Rvar(e_p)$ , then  $G \circ \langle e_p \rangle \in C$ .
- (P3) If  $H \circ \langle e_p \rangle \in C$ ,  $G \in C$ ,  $G|p = H|p$ , then  $G \circ \langle e'_p \rangle \in C$  for some event  $e'_p$  such that  $Rvar(e'_p) = Rvar(e_p)$  and  $Wvar(e'_p) = Wvar(e_p)$ .
- (P4) For any  $H \in C$ ,  $H \circ \langle e_p \rangle \in C$  implies that  $\alpha = value(v, H)$  holds, for all  $(v, \alpha) \in R$ , where  $e_p = [R, W, p]$ .

For notational simplicity, we make the following assumption, which requires each remote event to be either an atomic read or an atomic write.

**Atomicity Assumption:** Each event of a process  $p$  may either read or write (but not both) at most one variable that is remote to  $p$ .  $\square$

As explained later, this assumption actually can be relaxed to allow comparison primitives.

*Mutual exclusion systems.* We now define a special kind of shared-memory system, namely mutual exclusion systems, which are our main interest.

A *mutual exclusion system*  $\mathcal{S} = (C, P, V)$  is a shared-memory system that satisfies the following properties. Each process  $p \in P$  has a local variable  $stat_p$  ranging over  $\{ncs, entry, exit\}$  and initially  $ncs$ .  $stat_p$  is accessed only by the events  $Enter_p = [\{\}, \{(stat_p, entry)\}, p]$ ,  $CS_p = [\{\}, \{(stat_p, exit)\}, p]$ , and  $Exit_p = [\{\}, \{(stat_p, ncs)\}, p]$ , and is updated only as follows: for all  $H \in C$ ,

- $H \circ \langle Enter_p \rangle \in C$  iff  $value(stat_p, H) = ncs$ ;
- $H \circ \langle CS_p \rangle \in C$  only if  $value(stat_p, H) = entry$ ;
- $H \circ \langle Exit_p \rangle \in C$  only if  $value(stat_p, H) = exit$ .

(Note that  $stat_p$  transits directly from *entry* to *exit*.)

In our proof, we only consider computations in which each process enters and then exits its critical section at most once. Thus, we henceforth assume that each computation contains at most one  $Enter_p$  event for each process  $p$ . The remaining requirements of a mutual exclusion system are as follows.

**Exclusion:** For all  $H \in C$ , if both  $H \circ \langle CS_p \rangle \in C$  and  $H \circ \langle CS_q \rangle \in C$  hold, then  $p = q$ .

**Progress (starvation freedom):** For all  $H \in C$ , if  $\text{value}(\text{stat}_p, H) \neq \text{ncs}$ , then there exists an  $X$ -computation  $G$  such that  $H \circ G \circ \langle e_p \rangle \in C$ , where  $X = \{q \in P \mid \text{value}(\text{stat}_q, H) \neq \text{ncs}\}$  and  $e_p$  is either  $CS_p$  (if  $\text{value}(\text{stat}_p, H) = \text{entry}$ ) or  $\text{Exit}_p$  (if  $\text{value}(\text{stat}_p, H) = \text{exit}$ ).  $\square$

*Cache-coherent systems.* On cache-coherent shared-memory systems, some remote variable accesses may be handled without causing interconnect traffic. Our lower-bound proof applies to such systems without modification. This is because we do not count every remote event, but only critical events, as defined below.

**Definition 1.** Let  $\mathcal{S} = (C, P, V)$  be a mutual exclusion system. Let  $e_p$  be an event in  $H \in C$ . Then, we can write  $H$  as  $F \circ \langle e_p \rangle \circ G$ , where  $F$  and  $G$  are subcomputations of  $H$ . We say that  $e_p$  is a critical event in  $H$  iff one of the following conditions holds:

**State transition event:**  $e_p$  is one of  $\text{Enter}_p$ ,  $CS_p$ , or  $\text{Exit}_p$ .

**Critical read:** There exists a variable  $v$ , remote to  $p$ , such that  $v \in \text{Rvar}(e_p)$  and  $F \mid p$  does not contain a read from  $v$ .

**Critical write:** There exists a variable  $v$ , remote to  $p$ , such that  $v \in \text{Wvar}(e_p)$  and  $\text{writer}(v, F) \neq p$ .  $\square$

Note that state transition events do *not* actually cause cache misses; these events are defined as critical events because this allows us to combine certain cases in the proofs that follow. A process executes only three transition events per critical-section execution, so this does not affect our asymptotic lower bound.

According to Definition 1, a remote read of  $v$  by  $p$  is critical if it is the first read of  $v$  by  $p$ . A remote write of  $v$  by  $p$  is critical if (i) it is the first write of  $v$  by  $p$  (which implies that either  $\text{writer}(v, F) = q \neq p$  holds or  $\text{writer}(v, F) = \perp \neq p$  holds); or (ii) some other process has written  $v$  since  $p$ 's last write of  $v$  (which also implies that  $\text{writer}(v, F) \neq p$  holds).

Note that if  $p$  both reads and writes  $v$ , then both its first read of  $v$  and first write of  $v$  are considered critical. Depending on the system implementation, the latter of these two events might not generate a cache miss. However, even in such a case, the first such event will always generate a cache miss, and hence at least half of all such critical reads and writes will actually incur real global traffic. Hence, our lower bound remains asymptotically unchanged for such systems.

In a *write-through* cache scheme, writes always generate a cache miss. With a *write-back* scheme, a remote write to a variable  $v$  may create a cached copy of  $v$ , so that subsequent writes to  $v$  do not cause cache misses. In Definition 1, if  $e_p$  is not the first write to  $v$  by  $p$ , then it is considered critical only if  $\text{writer}(v, F) = q \neq p$  holds, which implies that  $v$  is stored in the local cache line of another process  $q$ . (Effectively, we are assuming an idealized cache of infinite size: a cached variable may be updated or invalidated but it is never replaced by another variable. Note that  $\text{writer}(v, F) = q$  implies that  $q$ 's cached copy of  $v$  has not been invalidated.) In such a case,  $e_p$  must either invalidate or update the cached copy of  $v$  (depending on the system), thereby generating global traffic.

Note that the definition of a critical event depends on the particular computation that contains the event, specifically the prefix of the computation preceding the event. Therefore, when saying that an event is (or is not) critical, the computation containing the event must be specified.

### 3 Proof Strategy

In Sec. 4 we show that for any positive  $k$ , there exists some  $N$  such that, for any mutual exclusion system  $\mathcal{S} = (C, P, V)$  with  $|P| \geq N$ , there exists a computation  $H$  such that some process  $p$  experiences point contention  $k$  and executes at least  $k$  critical events to enter and exit its critical section. The proof focuses on a special class of computations called “regular” computations. A regular computation consists of events of two groups of processes, “active processes” and “finished processes.” Informally, an active process is a process in its entry section, competing with other active processes; a finished process is a process that has executed its critical section once, and is in its noncritical section. (These properties follow from (R4), given later in this section.)

**Definition 2.** Let  $\mathcal{S} = (C, P, V)$  be a mutual exclusion system, and  $H$  be a computation in  $C$ . We define  $\text{Act}(H)$ , the set of active processes in  $H$ , and  $\text{Fin}(H)$ , the set of finished processes in  $H$ , as follows.

$$\begin{aligned} \text{Act}(H) &= \{p \in P \mid H \mid p \neq \langle \rangle \text{ and } \langle \text{Exit}_p \rangle \text{ is not in } H\} \\ \text{Fin}(H) &= \{p \in P \mid H \mid p \neq \langle \rangle \text{ and } \langle \text{Exit}_p \rangle \text{ is in } H\} \end{aligned}$$

□

The proof proceeds by inductively constructing longer and longer regular computations, until the desired lower bound is attained. The regularity condition defined below ensures that *no participating process has knowledge of any other process that is active*. This has two consequences: (i) we can “erase” any active process (*i.e.*, remove its events from the computation) and still get a valid computation; (ii) “most” active processes have a “next” critical event. In the definition that follows, (R1) ensures that active processes have no knowledge of each other; (R2) and (R3) bound the number of possible conflicts caused by appending a critical event; (R4) ensures that the active and finished processes behave as explained above; (R5) ensures that the property of being a critical write is conserved when considering certain related computations.

**Definition 3.** Let  $\mathcal{S} = (C, P, V)$  be a mutual exclusion system, and  $H$  be a computation in  $C$ . We say that  $H$  is regular iff the following conditions hold.

- (R1) For any event  $e_p$  and  $f_q$  in  $H$ , where  $p \neq q$ , if  $p$  writes to a variable  $v$ , and if another process  $q$  reads that value from  $v$ , then  $p \in \text{Fin}(H)$ .
- (R2) If a process  $p$  accesses a variable that is local to another process  $q$ , then  $q \notin \text{Act}(H)$ .
- (R3) For any variable  $v$ , if  $v$  is accessed by more than one processes in  $\text{Act}(H)$ , then either  $\text{writer}(v, H) = \perp$  or  $\text{writer}(v, H) \in \text{Fin}(H)$  holds.

- (R4) For any process  $p$  that participates in  $H$  ( $H \mid p \neq \langle \rangle$ ),  $\text{value}(\text{stat}_p, H)$  is entry, if  $p \in \text{Act}(H)$ , and ncs otherwise (i.e.,  $p \in \text{Fin}(H)$ ). Moreover, if  $p \in \text{Fin}(H)$ , then the last event of  $p$  in  $H$  is  $\text{Exit}_p$ .
- (R5) Consider two events  $e_p$  and  $f_p$  such that  $e_p$  precedes  $f_p$  in  $H$ , both  $e_p$  and  $f_p$  write to a variable  $v$ , and  $f_p$  is a critical write to  $v$  in  $H$ . In this case, there exists a write to  $v$  by some process  $r$  in  $\text{Fin}(H)$  between  $e_p$  and  $f_p$ .  $\square$

*Proof overview.* Initially, we start with a regular computation  $H_1$ , where  $\text{Act}(H_1) = P$ ,  $\text{Fin}(H_1) = \{\}$ , and each process has exactly one critical event. We then inductively show that other longer computations exist, the last of which establishes our lower bound. Each computation is obtained by rolling some process forward to its noncritical section (NCS) or by erasing some processes — this basic proof strategy has been used previously to prove several other lower bounds for concurrent systems [24, 7, 12]. We assume that  $P$  is large enough to ensure that enough non-erased processes remain after each induction step for the next step to be applied. The precise bound on  $|P|$  is given in Theorem 2.

At the  $j^{\text{th}}$  induction step, we consider a computation  $H_j$  such that  $\text{Act}(H_j)$  consists of  $n$  processes that execute  $j$  critical events each. We construct a regular computation  $H_{j+1}$  such that  $\text{Act}(H_{j+1})$  consists of  $\Omega(\sqrt{n}/k)$  processes that execute  $j+1$  critical events each. The construction method, formally described in Lemma 4, is explained below. In constructing  $H_{j+1}$  from  $H_j$ , some processes may be erased and *at most one* rolled forward. At the end of step  $k-1$ , we have a regular computation  $H_k$  in which each active process executes  $k$  critical events and  $|\text{Fin}(H_k)| \leq k-1$ . Since active processes have no knowledge of each other, a computation involving at most  $k$  processes can be obtained from  $H_k$  by erasing all but one active process; the remaining process performs  $k$  critical events.

We now describe how  $H_{j+1}$  is constructed from  $H_j$ . We show in Lemma 3 that, among the  $n$  processes in  $\text{Act}(H_j)$ , at least  $n-1$  can execute an additional critical event prior to its critical section. We call these events “future” critical events, and denote the corresponding set of processes by  $Y$ . We consider two cases, based on the variables remotely accessed by these future critical events.

**Erasing strategy.** Assume that  $\Omega(\sqrt{n})$  distinct variables are remotely accessed by the future critical events. For each such variable  $v$ , we select one process whose future critical event accesses  $v$ , and erase the rest. Let  $Y'$  be the set of selected processes. We now eliminate any information flow among processes in  $Y'$  by constructing a “conflict graph”  $\mathcal{G}$  as follows.

Each process  $p$  in  $Y'$  is considered a vertex in  $\mathcal{G}$ . By induction, process  $p$  has  $j$  critical events in  $\text{Act}(H_j)$  and one future critical event. An edge  $(p, q)$ , where  $p \neq q$ , is included in  $\mathcal{G}$  (i) if the future critical event of  $p$  remotely accesses a local variable of process  $q$ , or (ii) if one of  $p$ ’s  $j+1$  critical events accesses the same variable as the future critical event of process  $q$ .

Since each process in  $Y'$  accesses a distinct remote variable in its future critical event, it is clear that each process generates at most one edge by rule (i) and at most  $j+1$  edges by rule (ii). By applying Turán’s theorem (Theorem 1), we can find a subset  $Z$  of  $Y'$  such that  $|Z| = \Omega(\sqrt{n}/j)$  and their critical events



do not conflict with each other. By retaining  $Z$  and erasing all other active processes, we can eliminate all conflicts. Thus, we can construct  $H_{j+1}$ .

**Roll-forward strategy.** Assume that the number of distinct variables that are remotely accessed by the future critical events is  $O(\sqrt{n})$ . Since there are  $\Theta(n)$  future critical events, there exists a variable  $v$  that is remotely accessed by future critical events of  $\Omega(\sqrt{n})$  processes. Let  $Y_v$  be the set of these processes. First, we retain  $Y_v$  and erase all other active processes. Let the resulting computation be  $H'$ . We then arrange the future critical events of  $Y_v$  by placing all writes before all reads. In this way, the only information flow among processes in  $Y_v$  is that from the “last writer” of  $v$  to all the subsequent readers (of  $v$ ). Let  $p_{\text{LW}}$  be the last writer. We then roll  $p_{\text{LW}}$  forward by generating a regular computation  $G$  from  $H'$  such that  $\text{Fin}(G) = \text{Fin}(H') \cup \{p_{\text{LW}}\}$ .

If  $p_{\text{LW}}$  executes at least  $k$  critical events before reaching its NCS, then the  $\Omega(k)$  lower bound easily follows. Therefore, we can assume that  $p_{\text{LW}}$  performs fewer than  $k$  critical events while being rolled forward. Each critical event of  $p_{\text{LW}}$  that is appended to  $H'$  may generate information flow only if it reads a variable  $v$  that is written by another process in  $H'$ . Condition (R3) guarantees that if there are multiple processes that write to  $v$ , the last writer in  $H'$  is not active. Because information flow from an inactive process is allowed, a conflict arises only if there is a single process that writes to  $v$  in  $H'$ . Thus, each critical event of  $p_{\text{LW}}$  conflicts with at most one process in  $Y_v$ , and hence can erase at most one process. (Appending a noncritical event to  $H'$  cannot cause any processes to be erased. In particular, if a noncritical remote read by  $p_{\text{LW}}$  is appended, then  $p_{\text{LW}}$  must have previously read the same variable. By (R3), if the last writer is another process, then that process is not active.)

Therefore, the entire roll-forward procedure erases fewer than  $k$  processes from  $\text{Act}(H') = Y_v$ . We can assume  $|P|$  is sufficiently large to ensure that  $\sqrt{n} > 2k$ . This ensures that  $\Omega(\sqrt{n})$  processes survive after the entire procedure. Thus, we can construct  $H_{j+1}$ .

## 4 Lower Bound for Systems with Read/Write Atomicity

In this section, we present our lower-bound theorem for systems satisfying the Atomicity Assumption. At the end of this section, we explain why the lower bound also holds for systems with comparison primitives. We begin by stating several lemmas. Lemma 1 states that we can safely “erase” any active process. Lemma 2 allows us to extend a computation by noncritical events. Lemma 3 is used to show that “most” active processes have a “next” critical event.

**Lemma 1.** *Consider a regular computation  $H$  in  $C$ . For any set  $Y$  of processes such that  $\text{Fin}(H) \subseteq Y$ , the following hold:  $H|Y \in C$ ,  $H|Y$  is regular,  $\text{Fin}(H|Y) = \text{Fin}(H)$ , and an event  $e$  in  $H|Y$  is a critical event iff it is also a critical event in  $H$ .*  $\square$

**Lemma 2.** Consider a regular computation  $H$  in  $C$ , and a set of processes  $Y = \{p_1, p_2, \dots, p_m\}$ , where  $Y \subseteq \text{Act}(H)$ . Assume that for each  $p_j$  in  $Y$ , there exists a  $p_j$ -computation  $L_{p_j}$ , such that  $H \circ L_{p_j} \in C$  and  $L_{p_j}$  has no critical events in  $H \circ L_{p_j}$ . Define  $L$  to be  $L_{p_1} \circ L_{p_2} \circ \dots \circ L_{p_m}$ . Then, the following hold:  $H \circ L \in C$ ,  $H \circ L$  is regular,  $\text{Fin}(H \circ L) = \text{Fin}(H)$ , and  $L$  has no critical events in  $H \circ L$ .  $\square$

**Lemma 3.** Let  $H$  be a regular computation in  $C$ . Define  $n = |\text{Act}(H)|$ . Then, there exists a subset  $Y$  of  $\text{Act}(H)$ , where  $n-1 \leq |Y| \leq n$ , satisfying the following: for each process  $p$  in  $Y$ , there exist a  $p$ -computation  $L_p$  and an event  $e_p$  of  $p$  such that

- $H \circ L_p \circ \langle e_p \rangle \in C$ ;
- $L_p$  contains no critical events in  $H \circ L_p$ ;
- $e_p \notin \{\text{Enter}_p, \text{CS}_p, \text{Exit}_p\}$ ;
- $e_p$  is a critical event of  $p$  in  $H \circ L_p \circ \langle e_p \rangle$ ;
- $H \circ L_p$  is regular;
- $\text{Fin}(H \circ L_p) = \text{Fin}(H)$ .

□

The next theorem by Turán [13] will be used in proving Lemma 4.

**Theorem 1 (Turán).** Let  $\mathcal{G} = (V, E)$  be an undirected graph, where  $V$  is a set of vertices and  $E$  is a set of edges. If the average degree of  $\mathcal{G}$  is  $d$ , then there exists an independent set<sup>1</sup> with at least  $\lceil |V|/(d+1) \rceil$  vertices.  $\square$

The following lemma provides the induction step of our lower-bound proof.

**Lemma 4.** Let  $\mathcal{S} = (C, P, V)$  be a mutual exclusion system,  $k$  be a positive integer, and  $H$  be a regular computation in  $C$ . Define  $n = |\text{Act}(H)|$ . Assume that  $n > 1$  and

- each process in  $\text{Act}(H)$  executes exactly  $c$  critical events in  $H$ . (1)

Then, one of the following propositions is true.

**(Pr1)** There exist a process  $p \in \text{Act}(H)$  and a computation  $F \in C$  such that

- $F \circ \langle \text{Exit}_p \rangle \in C$ ;
- $F$  does not contain  $\langle \text{Exit}_p \rangle$ ;
- at most  $m$  processes participate in  $F$ , where  $m = |\text{Fin}(H) + 1|$ ;
- $p$  executes at least  $k$  critical events in  $F$ .

**(Pr2)** There exists a regular computation  $G$  in  $C$  such that

- $\text{Act}(G) \subseteq \text{Act}(H)$ ;
- $|\text{Fin}(G)| \leq |\text{Fin}(H) + 1|$ ;
- $|\text{Act}(G)| \geq \min(\sqrt{n}/(2c+3), \sqrt{n} - k)$ ;
- each process in  $\text{Act}(G)$  executes exactly  $c+1$  critical events in  $G$ . (2)

*Proof.* Because  $H$  is regular, using Lemma 3, we can construct a subset  $Y$  of  $\text{Act}(H)$  such that

$$n - 1 \leq |Y| \leq n, \tag{3}$$

and for each  $p \in Y$ , there exist a  $p$ -computation  $L_p$  and an event  $e_p$  such that

<sup>1</sup> An independent set of a graph  $\mathcal{G} = (V, E)$  is a subset  $V' \subseteq V$  such that no edge in  $E$  is incident to two vertices in  $V'$ .

- $H \circ L_p \circ \langle e_p \rangle \in C$ ; (4)
- $L_p$  contains no critical events in  $H \circ L_p$ ; (5)
- $e_p \notin \{Enter_p, CS_p, Exit_p\}$ ; (6)
- $e_p$  is a critical event of  $p$  in  $H \circ L_p \circ \langle e_p \rangle$ ; (7)
- $H \circ L_p$  is regular; (8)
- $\text{Fin}(H \circ L_p) = \text{Fin}(H)$ . (9)

Define  $V_{\text{fut}}$  as the set of variables remotely accessed by the “future” critical events:

$$V_{\text{fut}} = \{v \in V \mid \text{there exists } p \in Y \text{ such that } e_p \text{ remotely accesses } v\}. \quad (10)$$

We consider two cases, depending on the size of  $V_{\text{fut}}$ .

**Case 1:**  $|V_{\text{fut}}| \geq \sqrt{n}$  (**erasing strategy**) By definition, for each variable  $v$  in  $V_{\text{fut}}$ , there exists a process  $p$  in  $Y$  such that  $e_p$  remotely accesses  $v$ . Therefore, we can arbitrarily select one such process for each variable  $v$  in  $V_{\text{fut}}$  and construct a subset  $Y'$  of  $Y$  such that

- if  $p \in Y'$ ,  $q \in Y'$  and  $p \neq q$ , then  $e_p$  and  $e_q$  access different remote variables, and (11)
- $|Y'| = |V_{\text{fut}}| \geq \sqrt{n}$ . (12)

We now construct a graph  $\mathcal{G} = (Y', E_{\mathcal{G}})$ , where each vertex is a process in  $Y'$ . To each process  $y$  in  $Y'$ , we apply the following rules.

- (G1) Let  $v \in V_{\text{fut}}$  be the variable remotely accessed by  $e_y$ . If  $v$  is local to a process  $z$  in  $Y'$ , then introduce edge  $(y, z)$ .
- (G2) For each critical event  $f$  of  $y$  in  $H$ , let  $v_f$  be the variable remotely accessed by  $f$ . If  $v_f \in V_{\text{fut}}$  and  $v_f$  is remotely accessed by event  $e_z$  for some process  $z \neq y$  in  $Y'$ , then introduce edge  $(y, z)$ .

Because each variable is local to at most one process, and since an event can access at most one remote variable, (G1) can introduce at most one edge per process. Since, by (11),  $y$  executes exactly  $c$  critical events in  $H$ , by (12), (G2) can introduce at most  $c$  edges per process.

Combining (G1) and (G2), at most  $c + 1$  edges are introduced per process. Thus, the average degree of  $\mathcal{G}$  is at most  $2(c + 1)$ . Hence, by Theorem 1, there exists an independent set  $Z \subseteq Y'$  such that

$$|Z| \geq |Y'|/(2c + 3) \geq \sqrt{n}/(2c + 3), \quad (13)$$

where the latter inequality follows from (12).

Next, we construct a computation  $G$ , satisfying (Pr2), such that  $\text{Act}(G) = |Z|$ . Let  $H' = H \mid (Z \cup \text{Fin}(H))$ . For each process  $z \in Z$ , (4) implies  $H \circ L_z \in C$ . Hence, by (8) and (9), and applying Lemma 1 with ‘ $H$ ’  $\leftarrow H \circ L_z$  and ‘ $Y$ ’  $\leftarrow Z \cup \text{Fin}(H)$ , we have the following:

- $H' \circ L_z \in C$  (which, by (P1), implies  $H' \in C$ ), and
- an event in  $H' \circ L_z$  is critical iff it is also critical in  $H \circ L_z$ . (14)

By (5), the latter also implies that  $L_z$  contains no critical events in  $H' \circ L_z$ .

Let  $m = |Z|$  and index the processes in  $Z$  as  $Z = \{z_1, z_2, \dots, z_m\}$ . Define  $L = L_{z_1} \circ L_{z_2} \circ \dots \circ L_{z_m}$ . By applying Lemma 2 with ‘ $H$ ’  $\leftarrow H'$  and ‘ $Y$ ’  $\leftarrow Z$ , we have the following:

- $H' \circ L \in C$ ,
  - $H' \circ L$  is regular, and
  - $L$  contains no critical events in  $H' \circ L$ .
- (15)

By the definition of  $H'$  and  $L$ , we also have

- for each  $z \in Z$ ,  $(H' \circ L) \mid z = (H \circ L_z) \mid z$ .
- (16)

Therefore, by (4) and Property (P3), for each  $z_j \in Z$ , there exists an event  $e'_{z_j}$ , such that

- $G \in C$ , where  $G = H' \circ L \circ E$  and  $E = \langle e'_{z_1}, e'_{z_2}, \dots, e'_{z_m} \rangle$ ;
- $Rvar(e'_{z_j}) = Rvar(e_{z_j})$ ,  $Wvar(e'_{z_j}) = Wvar(e_{z_j})$ , and  $owner(e'_{z_j}) = owner(e_{z_j}) = z_j$ .

Conditions (R1)–(R5) can be individually checked to hold in  $G$ , which implies that  $G$  is a regular computation. Since  $Z \subseteq Y' \subseteq Y \subseteq \text{Act}(H)$ , by (11), (14), and (15), each process in  $Z$  executes exactly  $c$  critical events in  $H' \circ L$ .

We now show that every event in  $E$  is critical in  $G$ . Note that, by (7),  $e_z$  is a critical event in  $H \circ L_z \circ \langle e_z \rangle$ . By (6),  $e_z$  is not a transition event. By (16), the events of  $z$  are the same in both  $H \circ L_z$  and  $H' \circ L$ . Thus, if  $e_z$  is a critical read or a “first” critical write in  $H \circ L_z \circ \langle e_z \rangle$ , then it is also critical in  $G$ . The only remaining case is that  $e_z$  writes some variable  $v$  remotely, and is critical in  $H \circ L_z \circ \langle e_z \rangle$  because of a write to  $v$  prior to  $e_z$  by another process not in  $G$ . However, (R5) ensures that in such a case there exists some process in  $\text{Fin}(H)$  that writes to  $v$  before  $e_z$ , and hence  $e_z$  is also critical in  $G$ .

Thus, we have constructed a computation  $G$  that satisfies the following:  $\text{Act}(G) = Z \subseteq \text{Act}(H)$ ,  $\text{Fin}(G) = \text{Fin}(H') = \text{Fin}(H)$  (from the definition of  $H'$ , and since  $L \circ E$  does not contain transition events),  $|\text{Act}(G)| \geq \sqrt{n}/(2c + 3)$  (from (13)), and each process in  $\text{Act}(G)$  executes exactly  $c + 1$  critical events in  $G$  (from the preceding paragraph). It follows that  $G$  satisfies (Pr2).

**Case 2:**  $|V_{\text{fut}}| \leq \sqrt{n}$  (**roll-forward strategy**) For each variable  $v_j$  in  $V_{\text{fut}}$ , define  $Y_{v_j} = \{p \in Y \mid e_p \text{ remotely accesses } v_j\}$ . By (3) and (10),  $|V_{\text{fut}}| \leq \sqrt{n}$  implies that there exists a variable  $v_j$  in  $V_{\text{fut}}$  such that  $|Y_{v_j}| \geq (n - 1)/\sqrt{n}$  holds. Let  $v$  be one such variable. Then, the following holds:

$$|Y_v| \geq (n - 1)/\sqrt{n} > \sqrt{n} - 1. \quad (17)$$

Define  $H' = H \mid (Y_v \cup \text{Fin}(H))$ . Using  $Y_v \subseteq Y \subseteq \text{Act}(H)$ , we also have

$$\text{Act}(H') = Y_v \subseteq \text{Act}(H) \wedge \text{Fin}(H') = \text{Fin}(H). \quad (18)$$

Because  $H$  is regular, by Lemma 1,

- $H' \in C$ , (19)
- $H'$  is regular, and (20)
- an event in  $H'$  is a critical event iff it is also a critical event in  $H$ . (21)

We index processes in  $Y_v$  from  $y_1$  to  $y_m$ , where  $m = |Y_v|$ , such that if  $e_{y_i}$  writes to  $v$  and  $e_{y_j}$  reads  $v$ , then  $i < j$  (i.e., future writes to  $v$  precede future reads from  $v$ ).

For each  $y \in Y_v$ , let  $F_y = (H \circ L_y) \mid (Y_v \cup \text{Fin}(H))$ . (4) implies  $H \circ L_y \in C$ . Hence, by (8), and applying Lemma 1 with ‘ $H' \leftarrow H \circ L_y$ ’ and ‘ $Y' \leftarrow Y_v \cup \text{Fin}(H)$ ’,

we have the following:  $F_y \in C$ , and an event in  $F_y$  is critical iff it is also critical in  $H \circ L_y$ . Since  $y \in Y_v$  and  $L_y$  is a  $y$ -computation, by the definition of  $H'$ ,  $F_y = H' \circ L_y$ . Hence, by (5), we have

- $H' \circ L_y \in C$ , and (22)

- $L_y$  does not have a critical event in  $H' \circ L_y$ . (23)

Define  $L = L_{y_1} \circ L_{y_2} \circ \dots \circ L_{y_m}$ . We now use Lemma 2, with ' $H' \leftarrow H'$ ' and ' $Y' \leftarrow Y_v$ '. The antecedent of the lemma follows from (18), (19), (20), (22), and (23). This gives us the following.

- $H' \circ L \in C$ , (24)

- $H' \circ L$  is regular, (25)

- $\text{Fin}(H' \circ L) = \text{Fin}(H)$ , and (26)

- $L$  contains no critical events in  $H' \circ L$ . (26)

By the definition of  $H'$  and  $L$ , we also have

- for each  $y \in Y_v$ ,  $(H' \circ L) \upharpoonright y = (H \circ L_y) \upharpoonright y$ . (27)

Therefore, by (4) and Property (P3), for each  $y_j \in Y_v$ , there exists an event  $e'_{y_j}$ , such that

- $\bar{G} \in C$ , where  $\bar{G} = H' \circ L \circ E$  and  $E = \langle e'_{y_1}, e'_{y_2}, \dots, e'_{y_m} \rangle$ ;
- $Rvar(e'_{y_j}) = Rvar(e_{y_j})$ ,  $Wvar(e'_{y_j}) = Wvar(e_{y_j})$ , and  $owner(e'_{y_j}) = owner(e_{y_j}) = y_j$ .

From (6) and (26), it follows that  $L \circ E$  does not contain any transition events. Moreover, by the definition of  $L$  and  $E$ ,  $(L \circ E) \upharpoonright p \neq \langle \rangle$  implies  $p \in Y_v$ , for each process  $p$ . Combining these assertions with (18), we have

$$\begin{aligned} \text{Act}(\bar{G}) &= \text{Act}(H' \circ L) = \text{Act}(H') = Y_v \quad \wedge \\ \text{Fin}(\bar{G}) &= \text{Fin}(H' \circ L) = \text{Fin}(H') = \text{Fin}(H). \end{aligned} \quad (28)$$

We now claim that each process in  $Y_v (= \text{Act}(\bar{G}))$  executes exactly  $c + 1$  critical events in  $\bar{G}$ . In particular, by (11), (18), (21), and (26), it follows that each process in  $Y_v$  executes exactly  $c$  critical events in  $H' \circ L$ . On the other hand, by (7),  $e_y$  is a critical event in  $H \circ L_y \circ \langle e_y \rangle$ . By (27), and using an argument that is similar to that at the end of Case 1, we can prove that each event  $e'_y$  in  $E$  is a critical event in  $\bar{G}$ .

Let  $p_{\text{LW}}$  be the last process to write to  $v$  in  $\bar{G}$  (if such a process exists). If  $p_{\text{LW}}$  does not exist or if  $p_{\text{LW}} \in \text{Fin}(\bar{G}) = \text{Fin}(H)$ , conditions (R1)–(R5) can be individually checked to hold in  $\bar{G}$ , which implies that  $\bar{G}$  is a regular computation. In this case, (17) and (28) imply that  $\bar{G}$  satisfies (Pr2).

Therefore, assume  $p_{\text{LW}} \in \text{Act}(\bar{G}) = Y_v$ . Define  $H_{\text{LW}} = (H' \circ L) \upharpoonright (\text{Fin}(H) \cup \{p_{\text{LW}}\})$ . By (24), (25), and applying Lemma 1 with ' $H' \leftarrow H' \circ L$ ' and ' $Y' \leftarrow \text{Fin}(H') \cup \{p_{\text{LW}}\}$ ', we have:  $H_{\text{LW}} \in C$ ,  $H_{\text{LW}}$  is regular, and  $\text{Act}(H_{\text{LW}}) = \{p_{\text{LW}}\}$ .

Since  $p_{\text{LW}}$  is the only active process in  $H_{\text{LW}}$ , by the Progress property, there exists a  $p_{\text{LW}}$ -computation  $F$  such that  $H_{\text{LW}} \circ F \circ \langle \text{Exit}_{p_{\text{LW}}} \rangle \in C$  and  $F$  contains exactly one  $CS_{p_{\text{LW}}}$  and no  $\text{Exit}_{p_{\text{LW}}}$ . If  $F$  contains  $k$  or more critical events in  $H_{\text{LW}} \circ F$ , then  $H_{\text{LW}} \circ F$  satisfies (Pr1). Therefore, we can assume that  $F$  contains at most  $k - 1$  critical events in  $H_{\text{LW}} \circ L$ . Let  $V_{\text{LW}}$  be the set of variables remotely accessed by these critical events.

If a process  $q$  in  $Y_v$  writes to a variable in  $V_{\text{LW}}$  in  $H'$ , it might generate information flow between  $q$  and  $p_{\text{LW}}$ . Therefore, define  $K$ , the set of processes to erase (or “kill”), as  $K = \{p \in Y_v - \{p_{\text{LW}}\} \mid p = \text{writer}(u, H') \text{ or } u \text{ is local to } p \text{ for some } u \in V_{\text{LW}}\}$ . (R2) ensures that each variable in  $V_{\text{LW}}$  introduces at most one process into  $K$ . Thus, we have  $|K| \leq |V_{\text{LW}}| \leq k - 1$ .

Define  $S$ , the “survivors,” as  $S = Y_v - K$ , and let  $H_S = (H' \circ L) \mid (\text{Fin}(H) \cup S)$ . By (24), and applying Lemma 1 with ‘ $H' \leftarrow H' \circ L$ ’ and ‘ $Y' \leftarrow \text{Fin}(H') \cup S$ ’, we have the following:  $H_S \in C$ ,  $H_S$  is regular, and  $\text{Act}(H_S) = S$ . By the definition of  $L$ , we also have  $H_S \mid y = (H' \circ L) \mid y$ , for each  $y \in S$ . Since every event in  $E$  accesses only local variables and  $v$ , by (P2), we have  $H_S \circ E \in C$ .

Note that (P2) also implies that the first event of  $F$  is  $e'_{p_{\text{LW}}}$ . Hence, we can write  $F$  as  $e'_{p_{\text{LW}}} \circ F'$ . Define  $G = H_S \circ E \circ F'$ . Note that  $(H_S \circ E) \mid p_{\text{LW}} = (H_{\text{LW}} \mid p_{\text{LW}}) \circ \langle e'_{p_{\text{LW}}} \rangle$ , and that events of  $F'$  cannot read any variable written by processes in  $S$  other than  $p_{\text{LW}}$  itself. Therefore, by inductively applying (P2) over the events of  $F'$ , we have  $G \in C$ .

Conditions (R1)–(R5) can be individually checked to hold in  $G$ , which implies that  $G$  is a regular computation such that  $\text{Fin}(G) = \text{Fin}(H) \cup \{p_{\text{LW}}\}$ . Moreover, by (17) and from  $|K| \leq k - 1$ , we have  $|\text{Act}(G)| = |S| \geq (\sqrt{n} - 1) - (k - 1) \geq \sqrt{n} - k$ . Thus,  $G$  satisfies (Pr2).  $\square$

**Theorem 2.** *Let  $N(k) = (2k + 1)^{2(2^k - 1)}$ . For any mutual exclusion system  $\mathcal{S} = (C, P, V)$  and for any positive number  $k$ , if  $|P| \geq N(k)$ , then there exists a computation  $H$  such that at most  $k$  processes participate in  $H$  and some process  $p$  executes at least  $k$  critical operations in  $H$  to enter and exit its critical section.*

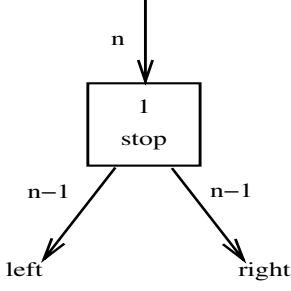
*Proof.* Let  $H_1 = \langle \text{Enter}_1, \text{Enter}_2, \dots, \text{Enter}_N \rangle$ , where  $P = \{1, 2, \dots, N\}$  and  $N \geq N(k)$ . By the definition of a mutual exclusion system,  $H_1 \in C$ . It is obvious that  $H_1$  is regular and each process in  $\text{Act}(H) = P$  has exactly one critical event in  $H_1$ . Starting with  $H_1$ , we repeatedly apply Lemma 4 and construct a sequence of computations  $H_1, H_2, \dots$  such that each process in  $\text{Act}(H_j)$  has  $j$  critical events in  $H_j$ . We repeat the process until either  $H_k$  is constructed or some  $H_j$  satisfies (Pr1) of Lemma 4.

If some  $H_j$  ( $j < k$ ) satisfies (Pr1), then consider the first such  $j$ . By our choice of  $j$ , each of  $H_1, \dots, H_{j-1}$  satisfies (Pr2) of Lemma 4. Therefore, since  $|\text{Fin}(H_1)| = 0$ , we have  $|\text{Fin}(H_j)| \leq j - 1 < k$ . It follows that computation  $F \circ \langle \text{Exit}_p \rangle$ , generated by applying Lemma 4 to  $H_j$ , satisfies Theorem 2.

The remaining possibility is that each of  $H_1, \dots, H_{k-1}$  satisfies (Pr2). We claim that, for  $1 \leq j \leq k$ , the following holds:

$$|\text{Act}(H_j)| \geq (2k + 1)^{2(2^{k+1-j} - 1)}. \quad (29)$$

The induction basis ( $j = 1$ ) directly follows from  $\text{Act}(H) = P$  and  $|P| \geq N(k)$ . In the induction step, assume that (29) holds for some  $j$  ( $1 \leq j < k$ ), and let  $n_j = |\text{Act}(H_j)|$ . Note that each active process in  $H_j$  executes exactly  $j$  critical events. By (29), we also have  $n_j > 4k^2$ , which implies that  $\sqrt{n_j} - k > \sqrt{n_j}/2 > \sqrt{n_j}/(2k + 1)$ . Therefore, by (2), we have  $|\text{Act}(H_{j+1})| \geq \min(\sqrt{n_j}/(2j + 3), \sqrt{n_j} - k) \geq \sqrt{n_j}/(2k + 1)$ , from which the induction easily follows.



(a)

**shared variable** $X: \{\perp\} \cup \{0..N-1\}$  initially  $\perp$ ; $Y$ : boolean initially *false***private variable** $dir$ :  $\{stop, left, right\}$ 

```

1:  $X := p$ ;
2: if  $Y$  then  $dir := left$ 
   else
3:    $Y := true$ ;
4:   if  $X = p$  then  $dir := stop$ 
5:   else  $dir := right$ 
   fi

```

(b)

**Fig. 1.** (a) The splitter element and (b) the code fragment that implements it.

Finally, (29) implies  $|\text{Act}(H_k)| \geq 1$ , and (Pr2) implies  $|\text{Fin}(H_k)| \leq k - 1$ . Hence, select any arbitrary process  $p$  from  $\text{Act}(H_k)$ . Define  $G = H_k \mid (\text{Fin}(H_k) \cup \{p\})$ . Clearly, at most  $k$  processes participate in  $G$ . By applying Lemma 1 with ' $H$ '  $\leftarrow H_k$  and ' $Y$ '  $\leftarrow \text{Fin}(H_k) \cup \{p\}$ , we have the following:  $G \in C$ , and an event in  $G$  is critical iff it is also critical in  $H_k$ . Hence, because  $p$  executes  $k$  critical events in  $H_k$ ,  $G$  is a computation that satisfies Theorem 2.  $\square$

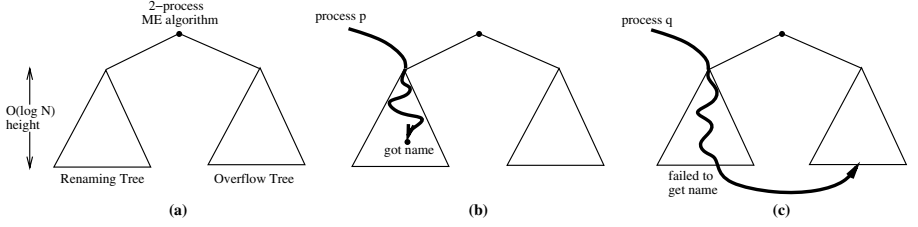
Theorem 2 can be easily strengthened to apply to systems in which comparison primitives are allowed. The key idea is this: if several comparison primitives on some shared variable are currently enabled, then they can be applied in an order in which at most one succeeds. A comparison primitive can be treated much like an ordinary write if successful, and like an ordinary read if unsuccessful.

## 5 Randomized Algorithm

In this section, we describe the randomized version of ALGORITHM AK mentioned earlier. Due to space constraints, only a high-level description of ALGORITHM AK is included here. A full description can be found in [3].

At the heart of ALGORITHM AK is the splitter element from Lamport's fast mutual exclusion algorithm [9]. The splitter element is defined in Fig. 1. Each process that invokes a splitter either stops or moves left or right (as indicated by the value assigned to the variable  $dir$ ). Splitters are useful because of the following properties: if  $n$  processes invoke a splitter, then at most one of them can stop at that splitter, and at most  $n - 1$  can move left (respectively, right).

In ALGORITHM AK, splitter elements are used to construct a "renaming tree." A splitter is located at each node of the tree and corresponds to a "name." A process acquires a name by moving down through the tree, starting at the root, until it stops at some splitter. Within the renaming tree is an arbitration



**Fig. 2.** (a) Renaming tree and overflow tree. (b) Process  $p$  gets a name in the renaming tree. (c) Process  $q$  fails to get a name and must compete within the overflow tree.

tree that forms dynamically as processes acquire names. A process competes within the arbitration tree by moving up to the root, starting at the node where it acquired its name. Associated with each node in the tree is a three-process mutual exclusion algorithm. As a process moves up the tree, it executes the entry section associated with each node it visits. After completing its critical section, a process retraces its path, this time executing exit sections. A three-process mutual exclusion algorithm is needed at each node to accommodate one process from each of the left- and right-subtrees beneath that node and any process that may have successfully acquired a name at that node.

In ALGORITHM AK, the renaming tree’s height is defined to be  $\lfloor \log N \rfloor$ , which results in a tree with  $\Theta(N)$  nodes. With a tree of this height, a process could “fall off” the end of the tree without acquiring a name. To handle such processes, a second arbitration tree, called the “overflow tree,” is used. The renaming and overflow trees are connected by placing a two-process mutual exclusion algorithm on top of each tree, as illustrated in Fig. 2.

The time complexity of ALGORITHM AK is determined by the depth to which a process descends in the renaming tree. If the point contention experienced by a process  $p$  is  $k$ , then the depth to which  $p$  descends is  $O(k)$ . This is because, of the processes that access the same splitter, all but one may move in the same direction from that splitter. If all the required mutual exclusion algorithms are implemented using Yang and Anderson’s local-spin algorithm [14], then because the renaming and overflow trees are both of height  $\Theta(\log N)$ , overall time complexity is  $O(\min(k, \log N))$ .

Our new randomized algorithm is obtained from ALGORITHM AK by replacing the original splitter with a probabilistic version, which is obtained by using “ $dir := \text{choice}(\text{left}, \text{right})$ ” in place of the assignments to  $dir$  at lines 2 and 5 in Fig. 1(b), where  $\text{choice}(\text{left}, \text{right})$  returns *left* (*right*) with probability  $1/2$ . With this change, a process descends to an expected depth of  $\Theta(\log k)$  in the renaming tree. Thus, the algorithm has  $\Theta(\log k)$  expected time complexity.

## 6 Concluding Remarks

We have established a lower bound that precludes an  $O(\log k)$  adaptive mutual exclusion algorithm (in fact, any  $o(k)$  algorithm) based on reads, writes, or com-



parison primitives. We have also shown that expected  $O(\log k)$  time is possible using randomization.

One may wonder whether a  $\Omega(\min(k, \log N / \log \log N))$  lower bound follows the results of this paper and [4]. Unfortunately, the answer is no. We have shown that  $\Omega(k)$  time complexity is required *provided*  $N$  is sufficiently large. This leaves open the possibility that an algorithm might have  $\Theta(k)$  time complexity for very “low” levels of contention, but  $o(k)$  time complexity for “intermediate” levels of contention. However, we find this highly unlikely.

## References

1. Y. Afek, H. Attiya, A. Fouren, G. Stupp, and D. Touitou. Long-lived renaming made adaptive. In *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing*, pages 91–103. May 1999.
2. Y. Afek, P. Boxer, and D. Touitou. Bounds on the shared memory requirements for long-lived and adaptive objects. In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing*, pages 81–89. July 2000.
3. J. Anderson and Y.-J. Kim. Adaptive mutual exclusion with local spinning. In *Proceedings of the 14th International Symposium on Distributed Computing*, pages 29–43, October 2000.
4. J. Anderson and Y.-J. Kim. An improved lower bound for the time complexity of mutual exclusion. To be presented at the 20th Annual ACM Symposium on Principles of Distributed Computing, August 2001.
5. J. Anderson and J.-H. Yang. Time/contention tradeoffs for multiprocessor synchronization. *Information and Computation*, 124(1):68–84, January 1996.
6. H. Attiya and V. Bortnikov. Adaptive and efficient mutual exclusion. In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing*, pages 91–100. July 2000.
7. J. Burns and N. Lynch. Mutual exclusion using indivisible reads and writes. In *Proceedings of the 18th Annual Allerton Conference on Communication, Control, and Computing*, pages 833–842, 1980.
8. M. Choy and A. Singh. Adaptive solutions to the mutual exclusion problem. *Distributed Computing*, 8(1):1–17, 1994.
9. L. Lamport. A fast mutual exclusion algorithm. *ACM Transactions on Computer Systems*, 5(1):1–11, February 1987.
10. M. Merritt and G. Taubenfeld. Speeding Lamport’s fast mutual exclusion algorithm. *Information Processing Letters*, 45:137–142, 1993.
11. E. Styer. Improving fast mutual exclusion. In *Proceedings of the 11th Annual ACM Symposium on Principles of Distributed Computing*, pages 159–168. August 1992.
12. E. Styer and G. Peterson. Tight bounds for shared memory symmetric mutual exclusion. In *Proceedings of the 8th Annual ACM Symposium on Principles of Distributed Computing*, pages 177–191. August 1989.
13. P. Turán. On an extremal problem in graph theory (in Hungarian). *Mat. Fiz. Lapok*, 48:436–452, 1941.
14. J.-H. Yang and J. Anderson. A fast, scalable mutual exclusion algorithm. *Distributed Computing*, 9(1):51–60, August 1995.

# Quorum-Based Algorithms for Group Mutual Exclusion<sup>\*</sup>

Yuh-Jzer Joung

Department of Information Management  
National Taiwan University  
Taipei, Taiwan  
joung@ccms.ntu.edu.tw

**Abstract.** We propose a quorum system, which we referred to as the *surficial* quorum system, for group mutual exclusion. The surficial quorum system is geometrically evident and so is easy to construct. It also has a nice structure based on which a truly distributed algorithm for group mutual exclusion can be obtained, and processes' loads can be minimized. When used with Maekawa's algorithm, the surficial quorum system allows up to  $\sqrt{\frac{2n}{m(m-1)}}$  processes to access a resource simultaneously, where  $n$  is the total number of processes, and  $m$  is the total number of groups. We also present two modifications of Maekawa's algorithm so that the number of processes that can access a resource at a time is not limited to the structure of the underlying quorum system, but to the number that the problem definition allows.

## 1 Introduction

*Group mutual exclusion* [16,20,4,13] is a generalization of mutual exclusion that allows a resource to be shared by processes of the same group, but requires processes of different groups to use the resource in a mutually exclusive style. As an application of the problem, assume that data is stored in a CD jukebox where only one disk can be loaded for access at a time. Then when a disk is loaded, users that need data on this disk can concurrently access the disk, while users that need a different disk have to wait until no one is using the currently loaded disk. Group mutual exclusion differs from *l-exclusion* [9] (which is also a generalization of mutual exclusion that allows at most  $l$  processes to be in the critical section simultaneously) in that in the latter the conflict in accessing a resource is due to the number of processes that attempt to access the resource, while in the former the conflict is due to the "type" of processes (i.e., the group they belong to).

---

<sup>\*</sup> Part of this research was done when the author was visiting Lab for Computer Science, Massachusetts Institute of Technology (1999-2000). Research supported in part by the National Science Council, Taipei, Taiwan, Grants NSC 89-2213-E-002-110.

Solutions for group mutual exclusion in shared memory models have been proposed in [16,20,4,13]. Here we consider message passing networks. Two message-passing solutions for group mutual exclusion have been proposed in [17]. Both are extensions of Ricart and Agrawala's algorithm for mutual exclusion [31]. Basically, they work as follows: a process wishing to enter a critical section (i.e., to use a shared resource) broadcasts a request to all processes in the system, and enters the critical section when all processes have acknowledged its request. Since all processes are involved in determining whether a process can enter the critical section, the algorithms cannot tolerate any single process failure.

In the literature, quorum systems have proven useful in coping with site failures and network partitions for both mutual exclusion (e.g., [11,26,2,21,8,1,30]), and  $l$ -exclusion (e.g., [19,23,24,28]). In general, a quorum system (called a *coterie* [11]) consists of a set of quora, each of which is a set of processes. Quora are used to guard the critical section. To enter the critical section, a process must *acquire* a quorum; that is, to obtain permission from every member of the quorum. Suppose that a quorum member gives permission to only one process at a time. Then mutual exclusion can be guaranteed by requiring every two quora in a coterie to intersect, and  $l$ -exclusion can be guaranteed by requiring any collection of  $l + 1$  quora to contain at least two intersecting quora. A quorum usually involves only a subset of the processes in the system. So even if processes may fail or become unreachable due to network partitions, some process may still be able to enter the critical section so long as not all quora are hit (a quorum is *hit* if some of its members fails).

It is easy to see that coteries for  $l$ -exclusion cannot be used for group mutual exclusion because two conflicting processes may then both enter the critical section after obtaining permissions from members of two disjoint quora respectively. On the other hand, coteries for mutual exclusion can be used for group mutual exclusion, but it will result in a degenerate solution in which only one process can be in the critical section at a time.

In this paper we present a quorum system, which we refer to as the *surficial* quorum system, for group mutual exclusion. To our knowledge, this is the first quorum system for group mutual exclusion to appear in the literature. The surficial quorum system is geometrically evident and so is easy to construct. It also has a nice structure based on which a truly distributed algorithm for group mutual exclusion can be obtained, and processes' loads can be minimized. When used with Maekawa's algorithm [26], the surficial quorum system allows up to  $\sqrt{\frac{2n}{m(m-1)}}$  processes to access a resource simultaneously, where  $n$  is the total number of processes, and  $m$  is the total number of groups. In contrast, only one process is allowed to access a resource at a time if an ordinary quorum system is used.

We also present a modification of Maekawa's algorithm so that the number of processes that can access a resource at a time is not limited to the structure of the underlying quorum system, but to the number that the problem definition allows. Thus, the modified algorithm can also use ordinary quorum systems to solve group mutual exclusion. Nevertheless, when used with our surficial quorum

system, the message complexity of the modified algorithm is still a factor of  $O(\sqrt{\frac{2n}{m(m-1)}})$  better than that used with an ordinary quorum system (of the same quorum size). Another modification that trades off synchronization delay for message complexity is also presented in the paper.

The rest of the paper is organized as follows. Section 2 gives the problem definition for group mutual exclusion. Section 3 presents the surficial quorum system. Section 4 presents quorum-based algorithms for group mutual exclusion. Conclusions and future work are offered in Section 5.

## 2 The Group Mutual Exclusion Problem

We consider a system of  $n$  asynchronous processes  $1, \dots, n$ , each of which cycles through the following three states, with *NCS* being the initial state:

- *NCS*: the process is outside CS (the *Critical Section*), and does not wish to enter CS.
- *trying*: the process wishes to enter CS, but has not yet entered CS.
- *CS*: the process is in CS.

The processes belong to  $m$  groups  $1, \dots, m$ . To make the problem more general, we do not require groups to be disjoint. When a process may belong to more than one group, the process must identify a unique group to which it belongs when it wishes to enter CS. Since group membership is concerned only at the time a process wishes to enter CS (and at the time the process is in CS), when we say ‘process  $i$  belongs to group  $j$ ’, we implicitly assume that process  $i$  has specified  $j$  as its group for entering CS.

The problem is to design an algorithm for the system satisfying the following requirements:

**mutual exclusion:** At any given time, no two processes of different groups are in CS simultaneously.

**lockout freedom:** A process wishing to enter CS will eventually succeed.

Moreover, to avoid degenerate solutions and unnecessary synchronization, we are looking for algorithms that can facilitate “**concurrent entering**”, meaning that if a group  $g$  of processes wish to enter CS and no other group of processes are interested in entering CS, then the processes in group  $g$  can concurrently enter CS [16, 20, 13].

## 3 A Quorum System for Group Mutual Exclusion

In this section we present a quorum system for group mutual exclusion. Let  $P = \{1, \dots, n\}$  be a set of nodes<sup>2</sup>, which belong to  $m$  groups. An ***m*-group**

<sup>1</sup> The problem is described in a more anthropomorphous setting as *Congenial Talking Philosophers* in [16].

<sup>2</sup> The terms *processes* and *nodes* will be used interchangeably throughout the paper. For a distinguishing purpose, however, we use “nodes” specifically to denote quorum members, and “processes” to denote group members.

**quorum system**  $\mathfrak{S} = (C_1, \dots, C_m)$  over  $P$  consists of  $m$  sets, where each  $C_i \subseteq 2^P$  is a set of subsets of  $P$  satisfying the following conditions:

**intersection:**  $\forall 1 \leq i, j \leq m, i \neq j, \forall Q_1 \in C_i, \forall Q_2 \in C_j : Q_1 \cap Q_2 \neq \emptyset$ .

**minimality:**  $\forall 1 \leq i \leq m, \forall Q_1, Q_2 \in C_i, Q_1 \neq Q_2 : Q_1 \not\subseteq Q_2$ .

We call each  $C_i$  a **cartel**, and each  $Q \in C_i$  a **quorum**.

Intuitively,  $\mathfrak{S}$  can be used to solve group mutual exclusion as follow: each process  $i$  of group  $j$ , when attempting to enter CS, must acquire permission from every member in a quorum  $Q \in C_j$ . Upon exiting CS, process  $i$  returns the permission to the members of the quorum. Suppose a quorum member gives permission to only one process at a time. Then, by the intersection property, no two processes of different groups can be in CS simultaneously. The minimality property is used rather to enhance efficiency. As is easy to see, if  $Q_1 \subseteq Q_2$ , then a process that can obtain permission from every member of  $Q_2$  can also obtain permission from every member of  $Q_1$ .

Recall that a quorum system over  $P$  for mutual exclusion is a set  $C \subseteq 2^P$  of quora satisfying the following requirements:

**intersection:**  $\forall Q_1, Q_2 \in C : Q_1 \cap Q_2 \neq \emptyset$ .

**minimality:**  $\forall Q_1, Q_2 \in C, Q_1 \neq Q_2 : Q_1 \not\subseteq Q_2$ .

To distinguish quorum systems for mutual exclusion from group quorum systems, we refer to the former as **ordinary** quorum systems.

An ordinary quorum system  $C$  can be used as an  $m$ -group quorum system by a straightforward transformation  $\mathfrak{T}_m$ :

$$\mathfrak{T}_m(C) = (C, \dots, C).$$

By the intersection property of  $C$ , all quora in a cartel of  $\mathfrak{T}_m(C)$  are pairwise intersected. Note that, in general, quora in the same cartel of a group quorum system need not intersect.

We define the **degree** of a cartel  $C$ , denoted as  $\deg(C)$ , to be the maximum number of pairwise disjoint quora in  $C$ . The **degree** of a group quorum system  $\mathfrak{S}$ , denoted as  $\deg(\mathfrak{S})$ , is the minimum  $\deg(C)$  among the cartels  $C$  in  $\mathfrak{S}$ . Clearly, if a node gives out its permission to at most one process at a time, then the number of processes (of the same group) that can be in CS simultaneously is limited to  $\deg(C)$ , where  $C$  is the cartel associated with the group. Moreover, a group quorum system of degree  $k$  immediately implies that every cartel contains at least an unhit quorum even if  $k-1$  processes have failed. So high degree group quorum systems also provide a better protection against faults.

In the following we present an  $m$ -group quorum system  $\mathfrak{S}_m = (C_1, \dots, C_m)$  with degree  $\sqrt{\frac{2n}{m(m-1)}}$ . In addition, the quora in the system satisfy the following four extra conditions:

1.  $\forall 1 \leq i, j \leq m : |C_i| = |C_j|$ .
2.  $\forall 1 \leq i, j \leq m, \forall Q_1 \in C_i, \forall Q_2 \in C_j : |Q_1| = |Q_2|$ .

3.  $\forall p, q \in P : |n_p| = |n_q|$ , where  $n_p$  is the multiset  $\{Q \mid \exists 1 \leq i \leq m : Q \in C_i \text{ and } p \in Q\}$ , and similar for  $n_q$ . In other words,  $|n_p|$  is the number of quora involving  $p$ .
4.  $\forall 1 \leq i, j \leq m, i \neq j, \forall Q_1 \in C_i, \forall Q_2 \in C_j : |Q_1 \cap Q_2| = 1$ .

Intuitively, the first condition ensures that each group has an equal chance in competing CS. The second condition ensures that the number of messages needed per entry to CS is independent of the quorum a process chooses. The third condition means that each node shares the same responsibility in the system. As argued by Maekawa [26], these three conditions are desirable for an algorithm to be truly distributed. The last condition simply minimizes the number of nodes that must be common to any two quora of different cartels, thereby reducing the size of a quorum.

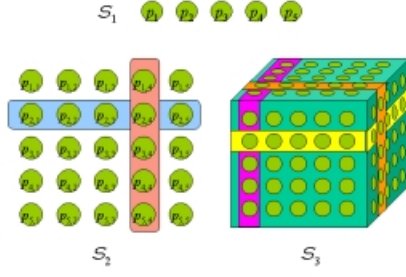
Before presenting the detailed construction of  $\mathfrak{S}_m$ , we first provide some intuitions. It is easy to see that a 1-group quorum system  $\mathfrak{S}_1$  satisfying the above conditions can be obtained as follows:

$$\mathfrak{S}_1 = (\{\{p\} \mid p \in P\})$$

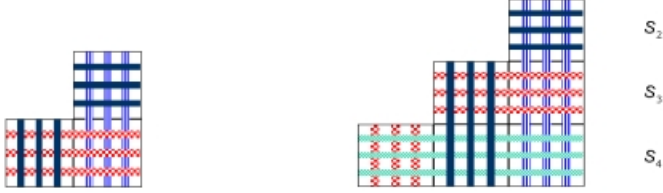
The quorum system can be viewed as a line consisting of  $n$  nodes, each of which corresponds to a process in  $P$ , where  $n = |P|$ . Each quorum then consists of exactly one node on the line, and the collection of the quora constitutes the only cartel in the system. See Figure 1 top. By extending this line to a two-dimensional plane, we can obtain a 2-group quorum system  $\mathfrak{S}_2 = (C_1, C_2)$ , where each quorum in  $C_1$  corresponds to the set of nodes in each row, while each quorum in  $C_2$  corresponds to the set of nodes in each column. By taking one step further, we can construct a 3-group quorum system  $\mathfrak{S}_3 = (C_1, C_2, C_3)$  by arranging nodes on the surface of a cube. Notice that a cube can be “wrapped up” by lines (strings) along three different dimensions. Lines along the same dimension are parallel to each other, while any two lines along different dimensions must intersect at two points. If we arrange the nodes on only three sides of the cube as shown in Figure 1, then every two lines along different dimensions intersect at exactly one point.

We can unfold the three sides of the cube on the plane as shown on the left of Figure 2. Each quorum in  $C_1$  then corresponds to a vertical line across the first (the right most) column of squares. Each quorum in  $C_2$  corresponds to a horizontal line across the top square, and a vertical line across the left square on the bottom. Finally, each quorum in  $C_3$  corresponds to a horizontal line across the two squares on the bottom.

By appending another three squares to the bottom of the above pile of squares and extending the lines to these extra squares, we can construct  $\mathfrak{S}_4$  as shown on the right of Figure 2. Each quorum in  $C_1$  corresponds to a vertical line across the first column of squares. Each quorum in  $C_2$  corresponds to a horizontal line across the square on the first level (starting from the top), and then a vertical line across the second column of squares. Each quorum in  $C_3$  corresponds to a horizontal line across the squares on the second level, and then a vertical line across the third column of squares. Finally, each quorum in  $C_4$  corresponds to



**Fig. 1.** A surficial quorum systems.



**Fig. 2.** The unfolded surficial quorum system for  $\mathfrak{S}_3$  (left), and  $\mathfrak{S}_4$  (right).

a horizontal line across the squares on the third level. Notice that on the right staircase of Figure 2 the first level of squares constitutes  $\mathfrak{S}_2$ , and the first two levels of squares constitutes  $\mathfrak{S}_3$ .

This procedure can be extended to  $\mathfrak{S}_m$ . In general, each  $C_i$  in  $\mathfrak{S}_m$  needs  $m - 1$  squares, each of which is to be shared with one of the other  $m - 1$  cartels so that the corresponding lines of the two cartels intersect at exactly one node on the square. Overall, there are  $m(m - 1)/2$  squares. Let  $k$  be the width of each square (i.e., the number of quora in each cartel). Then each square consists of  $k^2$  nodes. So the total number of nodes on the  $m(m - 1)/2$  squares is  $k^2 m(m - 1)/2$ . A simple way to map nodes on the squares to the processes in  $P$  is to let each node correspond to a unique process. In this case  $k^2 m(m - 1)/2 = n$ , where  $n = |P|$ . So

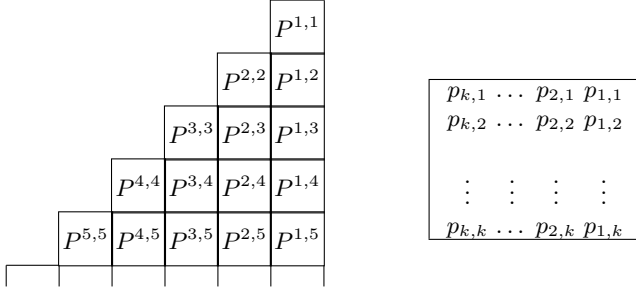
$$k = \sqrt{\frac{2n}{m(m - 1)}}, \quad m > 1.$$

So the quorum size is

$$(m - 1) \cdot k = \sqrt{\frac{2n(m - 1)}{m}}$$

and each cartel consists of  $\sqrt{\frac{2n}{m(m - 1)}}$  quora.

In the following we present the “staircase” construction of the surficial quorum system.



**Fig. 3.** Arrangement of nodes in  $\mathfrak{S}_m$ . On the left is the indices (superscripts) of squares, and on the right is the indices (subscripts) of nodes in each square.

**Inputs.**  $P \subseteq N$ ,  $n, m \in N$ , where  $n = |P|$ .

**Outputs.** An  $m$ -group quorum system  $\mathfrak{S}_m = (C_1, \dots, C_m)$  over  $P$ .

**Assumption.**  $m > 1$  and  $\sqrt{\frac{2n}{m(m-1)}} = k$  for some integer  $k$ .

1. Partition  $P$  into  $\frac{m(m-1)}{2}$  subsets, each of which consists of  $k^2$  nodes. Let  $P^{i,j}$  denote these subsets,  $1 \leq i \leq m-1$ ,  $1 \leq j \leq m-1$ . For each  $P^{i,j}$ , let  $p_{r,s}^{i,j}$  denote the nodes in the set, where  $1 \leq r, s \leq k$ . (See Figure 3.)
2. For each cartel  $C_i$  in  $\mathfrak{S}_m$ , denote the quora in the cartel by  $Q_{i,j}$ ,  $1 \leq i \leq m$ ,  $1 \leq j \leq k$ . Then,  $Q_{i,j}$  is defined by

$$Q_{i,j} = \{p_{r,j}^{s,i-1} | 1 \leq s \leq i-1, 1 \leq r \leq k\} \cup \{p_{j,r}^{i,s} | i \leq s \leq m-1, 1 \leq r \leq k\}$$

Note that when  $i = 1$ , the first set in the formula is empty, and when  $i = m$ , the second part is empty.

## Remarks

Some comments on the surficial quorum system are given in order. First, the degree of  $\mathfrak{S}_m$  is  $\sqrt{\frac{2n}{m(m-1)}}$ . In terms of degree, the construction is not optimal, as we can construct an  $m$ -group quorum system with degree  $\sqrt{n}$  [18]. It can be seen that  $\sqrt{n}$  is the upper bound as no  $m$ -group quorum system over a set of  $n$  nodes can have degree greater than  $\sqrt{n}$  for any  $m > 1$ . (This is because every quorum in an  $m$ -group quorum system of degree  $k$  must contain at least  $k$  elements. So every cartel must involve at least  $k \cdot k \leq n$  different nodes.) However, to reach such degree,  $n$  must be equal to some  $p^{2c}$ , where  $p$  is a prime and  $c$  is a positive integer. Besides, the construction is difficult to visualize as it works on an affine plane of order  $p^c$ . In contrast, the surficial quorum system is easy



			8 6 1
			9 4 2
			7 5 3
	5 9 1	9 5 1	
	7 2 6	7 6 2	
	3 4 8	8 4 3	
4 7 1	7 4 1	7 4 1	
8 2 5	5 2 8	8 5 2	
3 6 9	3 9 6	9 6 3	

$\mathfrak{S} = (C_1, C_2, C_3, C_4)$  where  
 $C_1 = \{\{1, 2, 3\}, \{4, 5, 6\}, \{7, 8, 9\}\}$   
 $C_2 = \{\{1, 6, 8\}, \{2, 4, 9\}, \{3, 5, 7\}\}$   
 $C_3 = \{\{3, 5, 9\}, \{2, 6, 7\}, \{3, 4, 8\}\}$   
 $C_4 = \{\{1, 4, 7\}, \{2, 5, 8\}, \{3, 6, 9\}\}$

**Fig. 4.** A mapping between processes and nodes and the resulting 4-group quorum system.

to visualize and so is easy to construct. Moreover, the surficial quorum system minimizes processes' loads by letting every node be included in at most two quora. Under this condition, the maximum degree an  $m$ -group quorum system can achieve is  $\sqrt{\frac{2n}{m(m-1)}}$ , which is exactly what  $\mathfrak{S}_m$  has achieved<sup>3</sup>.

Second, in the construction we have chosen the mapping between nodes and processes to be one-to-one. The advantage of this mapping is that it is simple and geometrically evident. The quorum size may be reduced (and therefore the increase of  $\deg(\mathfrak{S}_m)$ ) by choosing a more sophisticated mapping to allow nodes to map to the same process. For example, the mapping shown in Figure 4 for  $m = 4$  and  $n = 9$  results in a 4-group quorum system that is optimal in degree (i.e., with maximum possible degree)<sup>4</sup>. However, finding a mapping that optimizes the degree for arbitrary  $n$  and  $m$  is considerably more difficult and remains a challenging future work.

Third, any nonempty mapping  $\varphi$  from the nodes to the processes results in an  $m$ -group quorum system, although it may not satisfy the four extra conditions discussed at the beginning of this section. For example, the mapping  $\varphi(p_{r,s}^{i,j}) = q$  for any fixed  $q \in P$ ,  $1 \leq i \leq m-1, i \leq j \leq m-1, 1 \leq r, s \leq k$  results in a singleton-based  $m$ -group quorum system. Moreover, assume instead that  $k = \sqrt{n}$ . Let the processes be arranged as a grid so that  $P = \{q_{r,s} \mid 1 \leq r, s \leq k\}$ .

<sup>3</sup> To see this, let  $\mathfrak{C} = (C_1, \dots, C_m)$  be a group quorum system of degree  $k$  over an  $n$ -set. Let  $P_{r,s}^{i,j}$  be the intersection of the  $r$ th quorum in  $C_i$  and the  $s$ th quorum in  $C_j$ ,  $i \neq j$ . By definition,  $P_{r,s}^{i,j} \neq \emptyset$ . Since every node is included in at most two quora,  $P_{r,s}^{i,j} \cap P_{r',s'}^{i',j'} = \emptyset$  if  $\{i, j\} \neq \{i', j'\}$  (i.e., the two *unordered* pairs are not equal) or  $(r, s) \neq (r', s')$  (i.e., the two *ordered* pairs are not equal). Given that  $1 \leq i \neq j \leq m$  and that each cartel contains at least  $k$  quora, there are  $\binom{m}{2}$  unordered pairs of  $i, j$ , and for each of them, at least  $k^2$  ordered pairs of  $(r, s)$  that can constitute a  $P_{r,s}^{i,j}$ . So  $|\bigcup_{i,j,r,s} P_{r,s}^{i,j}| \geq \binom{m}{2} k^2$ . Since  $\binom{m}{2} k^2 \leq n$ , we have  $k \leq \sqrt{\frac{2n}{m(m-1)}}$ .

<sup>4</sup> Alternatively, the quorum system can be visualized as follows: take any one of the six squares. Then, any vertical line of three nodes constitutes a quorum, and the three vertical lines on the square constitute a cartel. Similarly, the three horizontal lines constitute a second cartel. The other two cartels correspond to the three “rounded” lines with slope 1 and  $-1$ , respectively. For example, the three “rounded” lines with slope 1 on the top square are:  $\{3, 6, 9\}$ ,  $\{5, 2, 8\}$ , and  $\{7, 4, 1\}$ .

Then the mapping  $\varphi(p_{r,s}^{i,j}) = q_{r,s}$  results in a quorum system  $\mathfrak{C} = (C_1, \dots, C_m)$  such that each quorum in  $C_1$  corresponds to a column in the grid, each quorum in  $C_m$  corresponds to a row in the grid, and each quorum in every other cartel  $C_i$  corresponds to a row and a column.<sup>5</sup> By experimenting different mappings and by adjusting the dimension of the squares (i.e., by replacing the squares with rectangles), we can fine-tune the quorum system to fit into different applications in which the demand of the shared resource by different groups of processes is off-balanced.

Finally, it is easy to see that  $\mathfrak{S}_m$  is in general “dominated.” Dominance between group quorum systems is defined as follows [18]:

**Definition 1.** Let  $\mathfrak{C} = (C_1, \dots, C_m)$  and  $\mathfrak{D} = (D_1, \dots, D_m)$  be two  $m$ -group quorum systems over  $P$ . Then  $\mathfrak{C}$  dominates  $\mathfrak{D}$  if

1.  $\mathfrak{C} \neq \mathfrak{D}$ ,
2.  $\forall Q \in D_i, 1 \leq i \leq m, \exists R \in C_i : R \subseteq Q$ .

$\mathfrak{D}$  is **nondominated** if there is no  $\mathfrak{C}$  such that  $\mathfrak{C}$  dominates  $\mathfrak{D}$ .

To illustrate dominance, the group quorum system  $\mathfrak{C} = (\{\{1, 2\}, \{3, 4\}\}, \{\{1, 3\}, \{2, 4\}\})$  over  $P = \{1, 2, 3, 4\}$  is dominated by  $\mathfrak{D} = (\{\{1, 2\}, \{3, 4\}, \{1, 4\}, \{2, 3\}\}, \{\{1, 3\}, \{2, 4\}\})$ . By a simple enumeration, it can be seen that  $\mathfrak{D}$  is nondominated.

Intuitively, a group quorum system  $C$  *dominates*  $D$  if whenever a quorum of a cartel in  $D$  can survive some failures, then some quorum in the corresponding cartel of  $C$  can certainly survive as well. Thus,  $C$  is said to be superior to  $D$  because  $C$  provides more protection against failures.

“Fully distributedness” and “nondominance” appear to be two conflicting notions, as for example, the “fully distributed” ordinary quorum system proposed by Maekawa [26] is also dominated [10][29]. However, the construction of  $\mathfrak{S}_m = (C_1, \dots, C_m)$  is such that every quorum  $Q$  in the cartels is a minimal set that intersects every other quorum in a different cartel. As proved in [18], this property implies that  $\mathfrak{S}_m$  can be extended to a nondominated group quorum system  $\mathfrak{U} = (D_1, \dots, D_m)$  such that  $C_i \subseteq D_i$  for all  $1 \leq i \leq m$ . In other words, we can build upon  $\mathfrak{S}_m$  a nondominated group quorum system  $\mathfrak{U}$  such that  $\mathfrak{U}$  “contains”  $\mathfrak{S}_m$ . An important meaning of this “containing” relation is that:  $\mathfrak{S}_m$  can be used to realize a truly distributed algorithm when failures do not occur, while  $\mathfrak{U}$  can be used to “backup”  $\mathfrak{S}_m$  when failures do occur to increase fault tolerance.

<sup>5</sup> Note that because in the construction a horizontal line of nodes is “connected” by a unique vertical line of nodes, not any union of a row and a column in the grid represents a quorum in  $C_i$ . However, by relaxing the “connection” condition, we can obtain a  $C_i$  that is composed of any union of a row and a column in the grid. Note further that  $C_1$  and  $C_m$  have degree  $\sqrt{n}$ , while the other cartels have degree 1.

## 4 Quorum-Based Algorithms for Group Mutual Exclusion

In this section we present two algorithms that use quorum systems to solve group mutual exclusion. The network is assumed to be complete and the communication channel between each pair of processes is reliable and FIFO. We begin with Maekawa's algorithm [26]. Then we discuss some variations involving trade-offs between concurrency and message complexity, and between concurrency and synchronization delay.

### 4.1 The Basic Framework

Most quorum-based algorithms for mutual exclusion are based on Maekawa's algorithm [26], which works as follows:

1. A process  $i$  wishing to enter CS sends a request message to each member of a quorum  $Q$ , and enters CS only after it has received a permission message from every member of  $Q$ .  
Upon leaving CS, a process sends a release message to every member of  $Q$  to release its permission.
2. A node gives away one permission at a time. So upon receipt of a request by  $i$ , a node  $j$  checks if it has given away its permission.
  - a) If  $j$  has given a permission message to another process  $k$ , and has not yet received a release message from  $k$ , then some arbitration mechanism is used to determine whether to let  $i$  wait, or to let  $i$  preempt  $k$ 's possession of the permission.
  - b) Otherwise,  $j$  sends a permission message to  $i$ .

In general, request messages by process  $i$  to the members of  $Q$  are sent simultaneously so as to minimize the *synchronization delay*, which is the delay from the time a process invokes a mutual exclusion request until the time it enters CS. The delay is measured in terms of message transmission time. It is clear that the minimum synchronization delay is 2 if request messages are sent simultaneously. However, due to the asynchrony of the system, a process may hold a permission while waiting for another process to release a permission. This in turn may incur a deadlock.

Maekawa's algorithm handles deadlocks by requiring low-priority processes to yield permissions to high-priority processes. Priorities are usually implemented by Lamport's logical timestamps [25]. The smaller the timestamp of a request, the higher the priority of the request. Specifically, if a node  $i$  receives a request by  $j$  after giving a permission to  $k$  and  $j$ 's priority is higher than  $k$ 's, then  $i$  issues an inquiry message to  $k$ . Process  $k$  then releases its permission to  $i$  (by sending a release message) if it determines that it cannot successfully acquire permissions from all members of the quorum it chooses. After receiving the release message, node  $i$  gives its permission to  $j$  by sending  $j$  a grant message. When  $j$  exits CS and releases  $i$ 's permission,  $i$  then returns the permission to  $k$  (presumably no

other process with a priority higher than  $k$ 's is also waiting for  $i$ 's permission). So Maekawa's algorithm needs  $3c$  to  $6c$  messages per entry to CS, where  $c$  is the (maximum) size of a quorum.

Maekawa's algorithm works straightforwardly for group mutual exclusion: Let  $\mathfrak{C} = (C_1, \dots, C_m)$  be an  $m$ -group quorum system over  $P$ . A process  $i \in P$  that wishes to enter CS as a member of group  $j$  chooses a quorum from the cartel  $C_j$ , and enters CS only when it has obtained a permission from every member of the quorum. By the mutual exclusion property of  $\mathfrak{C}$  and by the conflict resolution scheme used in the algorithms, mutual exclusion and lockout freedom are easily guaranteed.

#### 4.2 A Tradeoff between Concurrency and Message Complexity

In Maekawa's algorithm, since a node gives permission to only one process at a time, the maximum number of processes of a group that can be in CS simultaneously is limited to the degree of the cartel associated with the group. So no concurrency is offered using group quorum systems  $\mathfrak{T}_m(C)$  derived from ordinary quorum systems  $C$  (which have degree one).

For group quorum systems with degree greater than one, they may still not be able to offer a satisfactory degree of concurrency. This is because the size of a group can be greater than  $\sqrt{|P|}$ . However, as discussed in Section 3, no  $m$ -group quorum system over  $P$  can have degree more than  $\sqrt{|P|}$ , unless  $m = 1$ .

To overcome this, we modify Maekawa's algorithm so that a node can give permission to more than one process. So even if quora in the same cartel may intersect, two or more processes may still enter CS simultaneously. The rule for a node  $k$  to determine whether to grant  $i$ 's request for permission is as follows:

$k$  grants  $i$ 's request so long as there is no *conflict*—i.e., no other process of a different group is also requesting/possessing  $k$ 's permission. Otherwise, conflicts are resolved as follows: a process  $i$  of group  $g$  yields  $k$ 's permission to another process  $j$  of group  $h$  if  $j$  has priority higher than all processes of group  $g$  that currently request/possess  $j$ 's permissions.

Note that because of the rule, requests are not processed strictly in First-Come-First-Served (FCFS) order. In particular, a node  $k$  may grant  $i$ 's request even if some other  $j$  of a different group is already waiting for  $k$ 's permission (regardless of whether  $j$ 's priority is higher than  $i$  or not.) The reason for this is to increase system performance. We shall explain this in more detail in the following section.

We refer to the algorithm as Maekawa\_M. Because the algorithm is fairly easy to design, we omit the detailed code of the algorithm in this extended abstract.

#### 4.3 A Tradeoff between Concurrency and Synchronization Delay

In the above algorithm, after a node  $i$  has given permission to  $k$  processes,  $i$  may have to withdraw its permission if it receives a higher priority request from a different group. Withdrawing a permission from a process results in three

messages: an inquiry message, a relinquish message, and eventually the return of the permission to the process. So in the worst case, a request to  $i$  may generate  $3k$  messages. So the maximum number of messages per entry to CS is  $3c + 3c \cdot r$ , where  $c$  is the quorum size, and  $r$  is the maximum number of permissions a node may give away at a time. To facilitate a maximum concurrency while minimizing message complexity, for each cartel  $C$ ,  $r$  should be limited to  $s/\deg(C)$ , where  $s$  is the size of the group that uses  $C$ . In this case, an entry to CS requires at most  $3c + 3c \cdot s/\deg(C)$ . Given that  $s$  is determined by the problem definition, the message complexity is roughly in proportion to  $c/\deg(C)$ . So the higher the degree of the underlying group quorum system, the lower the message complexity.

If message complexity needs to be bounded in  $O(c)$ , deadlocks must be resolved in a different way. A well-known technique in resource allocation is to let each process request permissions from quorum members in some fixed order [14], say, with increasing node IDs. So if a process  $i$  is waiting for  $j$ 's permission, then every permission  $i$  holds must be from some  $f$  such that  $f < j$ . Moreover, every process  $k$  that currently holds  $j$ 's permission has either obtained permissions from all members of its quorum, or is waiting for a permission from some  $h$  such that  $h > j$ . So deadlocks are not possible because there is no circular waiting.

Note that the above deadlock free argument does not depend on how many permissions a node may give out at a time. That is, a node can still give out multiple permissions. The number of messages required per entry to CS is  $3c$ , and the minimum synchronization delay is  $2c$ . The message complexity and the minimum synchronization delay can be reduced further to  $2c + 1$  and  $c + 1$ , respectively, by letting quorum members circulate request messages. The complete code of the algorithm is given in Figures 5 and 6. We refer to the algorithm as Maekawa.S. For ease of understanding, the algorithm is presented as two CSP-like repetitive commands consisting of guarded commands [15]: Figure 5 describes the behavior of a process that acts as a group member, and Figure 6 describes the behavior of a node that acts as a quorum member. If one wishes, the two repetitive commands can be combined into a single one.

Although we have assumed FIFO communication channels, in the algorithm a node  $j$  may receive a process  $i$ 's request before it receives  $i$ 's release message for  $i$ 's previous request (lines D.9-10). This occurs because request messages hop through quorum members. So when  $i$  issues a release message  $msg_1$  to  $j$  and then issues a new request  $msg_2$ ,  $msg_2$  may arrive at  $j$  (indirectly through members of a different quorum) before  $msg_1$  does. To simplify the algorithm,  $j$  defers the process of  $msg_2$  until it has received  $msg_1$  (lines E.16-18).

Like Maekawa\_M, requests are not processed strictly in FCFS order. Instead, when a node  $j$  receives a request from a process  $i$  of group  $g$ , if  $j$  has no outstanding permission, then  $j$  sends  $i$  a permission and chooses  $i$  as a reference (line D.7). A reference process is used such that subsequent requests from the same group are also granted so long as the reference remains in CS. When a reference process leaves CS, if no other process of a different group is waiting for  $j$ 's permission, then a new reference is chosen from those processes that currently hold  $j$ 's permissions (lines E.3-6). Otherwise, the reference is reset to  $\perp$ ,

```

A.1  * $[$  wish to enter CS as a member of group  $g \rightarrow$ 
A.2     $state := trying;$ 
A.3    randomly select a quorum  $Q$  from  $C_g$ ;
A.4    send REQUEST( $i, g, Q$ ) to  $first(Q)$ ;
B.1   $\square$  receive GRANT( $j$ )  $\rightarrow$ 
B.2     $state := CS;$  /* acquires quorum  $Q$  */
C.1   $\square$  exit CS  $\rightarrow$ 
C.2     $state := NCS;$ 
C.3    for  $j \in Q$  do send RELEASE( $i$ ) to  $j$ ;
C.4   $]$ 

```

#### Variables:

- $state$ : the state of process  $i$ .
- $Q$ : the quorum  $i$  selects. We assume the following three functions on quora:
  - $first(Q)$ : return the smallest id in  $Q$ .
  - $next(k, Q)$ : return the smallest id in  $Q$  that is greater than  $k$ .
  - $last(Q)$ : return the largest id in  $Q$ .
- $C_g$ : the cartel associated with group  $g$ .

#### Messages:

- REQUEST( $i, g, Q$ ): a request by  $i$  to obtain the recipient's permission to enter CS as a member of group  $g$ .  $Q$  is the (id of the) quorum  $i$  chooses.
- GRANT( $j$ ): a permission given by node  $j$  to the recipient. In particular,  $j$  is the last node in  $Q$ .
- RELEASE( $i$ ): a message by  $i$  to release the recipient's permission.

**Fig. 5.** Algorithm Maekawa.S executed by process  $i$ .

meaning that the “door” to CS for the group is closed to yield the opportunity to another group.

There are two reasons for choosing this “entry policy”. First, by the mutual exclusion property, while some reference process  $i$  is in CS, no other group of processes can be in CS. So maximal resource utilization can be achieved by allowing more processes of the same group to share CS with  $i$ , regardless of whether some other group of processes are waiting for CS or not. Furthermore, because while  $i$  is in CS, some fast process may enter and exit CS any number of times, the algorithm facilitates an unbounded degree of concurrency [16]. Note that lockout freedom can still be guaranteed because a reference process will eventually leave CS and close the “door” to CS for its group.

The other reason is to minimize the number of “context switches” [17]. (A *context switch* occurs when the next entry to CS is by a different group of process.) As analyzed in [17], in group mutual exclusion requests to CS cannot be processed in a strictly FCFS order, or else the system could degenerate to the case in which nearly only one process can be in CS at a time when  $m$  is large. As can be seen, this phenomenon will also appear in both Maekawa.M

```

D.1  *[ receive REQUEST( $i, g, Q$ )  $\longrightarrow$ 
D.2    if  $grant\_ps = \emptyset \vee (g = current\_group \wedge reference \neq \perp \wedge i \notin grant\_ps)$  then [
D.3      /* grant the request */
D.4      if  $j = last(Q)$  then send GRANT( $j$ ) to  $i$ ;
D.5      else send REQUEST( $i, g, Q$ ) to  $next(j, Q)$ ;
D.6      if  $grant\_ps = \emptyset$  then [
D.7         $current\_group := g$ ;
D.8         $reference := i$ ; ]
D.9       $grant\_ps := grant\_ps + \{i\}$ ; ]
D.10   else if  $i \in grant\_ps$  /*  $i$ 's request arrives before its previous release message */
D.11    $early\_requests := early\_requests + \{ \langle i, g, Q \rangle \}$ ;
D.12   else  $request\_ps := request\_ps + \{ \langle i, g, Q \rangle \}$ ; ]
E.1  □ receive RELEASE( $i$ )  $\longrightarrow$ 
E.2     $grant\_ps := grant\_ps - \{i\}$ ;
E.3    if  $reference = i$  then
E.4      if  $grant\_ps \neq \emptyset \wedge request\_ps = \emptyset$  then /* choose a new reference */
E.5         $reference := k$  for some arbitrary  $k \in grant\_ps$ ;
E.6      else  $reference := \perp$ ;
E.7    if  $grant\_ps = \emptyset \wedge request\_ps \neq \emptyset$  then [
E.8      /* grant the earliest request in the queue, as well as the requests from
E.9      the same group of processes */
E.10      $\langle k, h, R \rangle := first(request\_ps)$ ;
E.11      $reference := k$ ;
E.12      $current\_group := h$ ;
E.13     for  $\langle k', h', R' \rangle \in request\_ps, h = h',$  do [
E.14       if  $j = last(Q)$  then send GRANT( $j$ ) to  $k'$ ;
E.15       else send REQUEST( $k', h', R'$ ) to  $next(j, R')$ ;
E.16        $request\_ps := request\_ps - \{ \langle k', h', R' \rangle \}$ ;
E.17        $grant\_ps := grant\_ps + \{k'\}$ ; ]
E.18   if  $\langle i, g, S \rangle \in early\_requests$  for some  $g$  and  $S$ , then [
E.19     /* process  $i$ 's early request */
E.20     send REQUEST( $i, g, S$ ) to  $j$ ;
E.21      $early\_requests := early\_requests - \{ \langle i, g, S \rangle \}$ ; ]
    ]

```

#### Variables:

- $request\_ps$ : queue of requests received by  $j$ . The requests are represented as tuples  $\langle i, g, Q \rangle$ , where  $i$  is the requester,  $g$  is the group of the requester, and  $Q$  is the quorum  $i$  chooses. The queue is initialized to  $\emptyset$ . Requests in the queue are ordered by the time they are inserted into the queue. Function  $first(request\_ps)$  returns the earliest request in the queue.
- $early\_requests$ : queue of “early” requests received by  $j$ . A request  $\langle i, g, Q \rangle$  is said “early” and is temporarily stored in  $early\_requests$  if  $i$  has released  $j$ 's permission before it issued the request, but the request arrives at  $j$  before the release message.
- $grant\_ps$ : set of processes to which  $j$  has given a permission. It is initialized to  $\emptyset$ .
- $current\_group$ : the group of the processes to which  $j$  has given permissions. It is initialized to  $\perp$ .
- $reference$ : a reference process used to determine whether subsequent processes of  $current\_group$  can enter CS.

**Fig. 6.** Algorithm Maekawa\_S executed by node  $j$ .

and Maekawa.S. So in both algorithms we allow some late requests to “jump over” existing requests to allow more processes to concurrently enter CS. Due to the space limitation, more detailed performance analysis will be presented in the full paper.

#### 4.4 Correctness of the Algorithms

Maekawa’s algorithm has been well studied in [26,32,33,6,7]. It is easy to see that the algorithm can be directly adapted to group mutual exclusion using group quorum systems. For the two modified algorithms Maekawa\_M and Maekawa\_S we have presented, their correctness can also be easily seen. So to save space, we will leave the formal analysis in the full paper.

### 5 Conclusions and Future Work

We have presented a quorum system, the surficial quorum system, for group mutual exclusion. The surficial quorum system generalizes existing quorum systems for mutual exclusion in that quora for processes of the same group need not intersect with one another. This generalization allows processes to acquire quora simultaneously, and so allows them to enter critical section concurrently. The surficial quorum system has a very simple geometrical structure, based on which a truly distributed algorithm for group mutual exclusion can be obtained, and based on which processes’ loads can be minimized.

The surficial quorum system has degree  $\sqrt{\frac{2n}{m(m-1)}}$ , where  $n$  is the total number of processes, and  $m$  is the total number of groups. So when used with Maekawa’s algorithm, it allows a maximum of  $\sqrt{\frac{2n}{m(m-1)}}$  processes to be in the critical section simultaneously. The message complexity per entry to the critical section is  $6 \cdot \sqrt{\frac{2n(m-1)}{m}}$ . Furthermore, it can tolerate up to  $\sqrt{\frac{2n}{m(m-1)}} - 1$  process failures. For comparison, the two message-passing algorithms RA1 and RA2 presented in [17] have message complexity  $2n$  and  $3n$ , respectively, and both allow all group members to be in the critical section simultaneously. However, they cannot tolerate any single process failure. In terms of minimum synchronization delay, all three algorithms have the same measure—2.

As we have noted earlier, the degree of a group quorum system is bounded by  $\sqrt{n}$ . So Maekawa’s algorithm must be generalized if group size is greater than  $\sqrt{n}$  and we wish to allow all group members to be in the critical section simultaneously. Two generalizations Maekawa\_M and Maekawa\_S were presented in the paper. Both allow all group members to be in the critical section simultaneously, regardless of the degree of the underlying quorum systems. Maekawa\_M preserves Maekawa’s minimum synchronization delay, but needs  $3c + 3c \cdot s/d$  messages per entry to the critical section (when a node gives away bounded number of permissions), where  $c$  is the quorum size,  $s$  is the group size, and  $d$  is the degree of the underlying group quorum system. So when used with the surficial quorum system, the message complexity is  $O(n \cdot \min\{m, n\})$ . The other algorithm



Maekawa\_S trades off minimum synchronization delay for message complexity. It reduces the message complexity to  $2c + 1$ , but needs a minimum synchronization delay of  $c + 1$ .

There is a considerable literature on quorum systems. Many structures have been explored to construct ordinary quorum systems, including *finite projective planes* [26], *weighted voting* [12, 113], *grids* [22, 81], *trees* [2], *wheels* [27], *walls* [30], and *planar graphs* [5]. The structure we used in the surficial quorum system can be viewed as a “multidimensional” grid. For future work, it is interesting to investigate how the other structures can be used for group quorum systems.

**Acknowledgments.** I would like to thank Nancy Lynch for giving me a valuable opportunity to visit her group, and to discuss this research with her, as well as with the other members of the TDS group, including Idit Keidar, Alex Shvartsman, and Victor Luchangco. I would also like to thank Michel Raynal for stimulating this research, and the anonymous referees for their useful comments.

## References

1. D. Agrawal, Ö. Egecioglu, and A. El Abbadi. Billard quorums on the grid. *IPL*, 64(1):9–16, 14 October 1997.
2. D. Agrawal and A. El Abbadi. An efficient and fault-tolerant solution for distributed mutual exclusion. *ACM TOCS*, 9(1):1–20, February 1991.
3. M. Ahamad and M. H. Ammar. Multidimensional voting. *ACM TOCS*, 9(4):399–431, November 1991.
4. K. Alagarsamy and K. Vidyasankar. Elegant solutions for group mutual exclusion problem. Technical report, Dept. of Computer Science, Memorial University of Newfoundland, Canada, 1999.
5. R. A. Bazzi. Planar quorums. *TCS*, 243(1–2):243–268, July 2000.
6. Y.-I. Chang. A correct  $O(\sqrt{N})$  distributed mutual exclusion algorithm. In *Proc. 5th Int'l. Conf. on Parallel and Distributed Computing and Systems*, pages 56–61, 1992.
7. Y.-I. Chang. Notes on Maekawa's  $O(\sqrt{N})$  distributed mutual exclusion algorithm. In *Proc. SPDP '93*, pages 352–359, 1994.
8. S. Y. Cheung, M. H. Ammar, and M. Ahamad. The grid protocol: A high performance scheme for maintaining replicated data. *IEEE TKDE*, 4(6):582–59, December 1992.
9. M. J. Fischer, N. A. Lynch, J. E. Burns, and A. Borodin. Resource allocation with immunity to limited process failure (preliminary report). In *Proc. 20th FOCS*, pages 234–254, 1979.
10. W. C. Ada Fu. *Enhancing concurrency and availability for database systems*. PhD thesis, Simon Fraser University, Burnaby, British Columbia, Canada, 1990.
11. H. Garcia-Molina and D. Barbara. How to assign votes in a distributed system. *JACM*, 32(4):841–860, October 1985.
12. David K. Gifford. Weighted voting for replicated data. In *Proc. 7th ACM SOSP*, pages 150–162, 1979.
13. V. Hadzilacos. A note on group mutual exclusion. In *Proc. 20th PODC*, 2001.
14. J. W. Havender. Avoiding deadlock in multiasking systems. *IBM Systems Journal*, 7(2):74–84, 1968.

15. C. A. R. Hoare. Communicating sequential processes. *CACM*, 21(8):666–677, August 1978.
16. Y.-J. Joung. Asynchronous group mutual exclusion (extended abstract). In *Proc. 17th PODC*, pages 51–60, 1998. Full paper in *Distributed Computing*, 13(4):189–206, 2000.
17. Y.-J. Joung. The congenial talking philosophers problem in computer networks (extended abstract). In *Proc. 13th DISC*, LNCS 1693, pages 195–209, 1999.
18. Y.-J. Joung. On generalized quorum systems. Technical report, Department of Information Management, National Taiwan University, Taipei, Taiwan, 2000.
19. H. Kakugawa, S. Fujita, M. Yamashita, and T. Ae. Availability of  $k$ -coterie. *IEEE Trans. on Computers*, 42(5):553–558, May 1993.
20. P. Keane and M. Moir. A simple local-spin group mutual exclusion algorithm. In *Proc. 18th PODC*, pages 23–32, 1999.
21. A. Kumar. Hierarchical quorum consensus: A new algorithm for managing replicated data. *IEEE Trans. on Computers*, 40(9):996–1004, September 1991.
22. A. Kumar and S. Y. Cheung. A high availability  $\sqrt{N}$  hierarchical grid algorithm for replicated data. *IPL*, 40(6):311–316, 30 December 1991.
23. Y.-C. Kuo and S.-T. Huang. A simple scheme to construct  $k$ -coterie with  $O(\sqrt{N})$  uniform quorum sizes. *IPL*, 59(1):31–36, 8 July 1996.
24. Y.-C. Kuo and S.-T. Huang. A geometric approach for constructing coterie and  $k$ -coterie. *IEEE TPDS*, 8(4):402–411, April 1997.
25. L. Lamport. Time, clocks and the ordering of events in a distributed system. *CACM*, 21(7):558–565, July 1978.
26. M. Maekawa. A  $\sqrt{N}$  algorithm for mutual exclusion in decentralized systems. *ACM TOCS*, 3(2):145–159, May 1985.
27. Y. Marcus and D. Peleg. Construction methods for quorum systems. Technical Report CS92-33, The Weizmann Institute of Science, Rehovot, Israel, 1992.
28. M. L. Neilsen. Properties of nondominated  $K$ -coterie. *The J. of Systems and Software*, 37(1):91–96, April 1997.
29. D. Peleg and A. Wool. The availability of quorum systems. *IEC*, 123(2):210–223, December 1995.
30. D. Peleg and A. Wool. Crumbling walls: A class of practical and efficient quorum systems. *DC*, 10(2):87–97, 1997.
31. G. Ricart and A. K. Agrawala. An optimal algorithm for mutual exclusion in computer networks. *CACM*, 24(1):9–17, January 1981.
32. B. A. Sanders. The information structure of distributed mutual exclusion algorithms. *ACM TOCS*, 5(3):284–299, August 1987.
33. M. Singhal. A class of deadlock-free Maekawa-type algorithms for mutual exclusion in distributed systems. *DC*, 4(3):131–138, 1991.

# An Effective Characterization of Computability in Anonymous Networks

Paolo Boldi and Sebastiano Vigna

Dipartimento di Scienze dell'Informazione, Università degli Studi di Milano, Italy

**Abstract.** We provide effective (i.e., recursive) characterizations of the relations that can be computed on networks where all processors use the same algorithm, start from the same state, and know at least a bound on the network size. Three activation models are considered (synchronous, asynchronous, interleaved).

## 1 Introduction

The question concerning which problems can be solved by a distributed system when all processors use the same algorithm and start from the same state has a long story: it was firstly formulated by Angluin [Ang80], who investigated the problem of establishing a “center”. She was the first to realize the connection with the theory of *graph coverings*, which was going to provide, in particular with the work of Yamashita and Kameda [YK96], several characterization for problems which are solvable under certain topological constraints. Further investigation led to the classification of computable functions [YK96,YK98,ASW88,Nor96], and allowed to eliminate several restrictions (such as bidirectionality, distinguished links, synchronicity, . . . ) [DKMP95,BCG<sup>+</sup>96,BV97a].

Few years ago, while lecturing at the Weizmann Institute about possibility and impossibility results for function computation [BV97a], we were asked by Moni Naor whether our results could be extended to the computation of arbitrary *relations* (as opposed to the undecidability results of [NS95]). Of course, this is the most general case of a distributed task: all classical problems such as election, topology reconstruction etc., can be seen as the computation of a specific relation.

The present paper answers positively to this question. We provide a *proof technique* that allows to show whether an anonymous algorithm that computes a given relation exists under a wide range of models. Moreover, the proof technique is *effective*: if the class of networks under examination is finite, and the relation to be computed is finite, then the technique turns into a recursive procedure that provides either an anonymous algorithm computing the relation, or a proof of impossibility.

Of course, results about specific relations (such as the one defining the election problem or topology reconstruction) are already known, at least for certain models. Here we complete the picture by providing a characterization of the relations that can be computed on a wide range of models when (as usually assumed) a bound on the network size is known. (If such a bound is not known, a completely different approach is needed, as shown in [BV99].)

Our results are mainly of theoretical interest, because of the large amount of information exchanged by the processors. Moreover, complexity issues are not taken into consideration, as opposed, for instance, to [ASW88, ANIM96]. We rather concentrate on general decidability properties that hold under any assumption of knowledge or of communication primitives (broadcast, point-to-point, etc.).

Our networks are just directed graphs coloured on their arcs (information such as processor identity, communication models etc. can be easily encoded on the arc colouring [BCG<sup>+</sup>96]), and each processor changes its state depending on its previous state and on the state of its in-neighbours. The problem specification is given by a certain relation between the inputs and the outputs of the processors, and we allow the relation to depend on the network, so that problems like topology reconstruction can be specified.

We consider three known activation models: asynchronous (at each step, we can activate any subset of enabled processors), synchronous (all processors) and interleaved (a.k.a. *central daemon*—exactly one processor), and we give characterization theorems based on the notion of *graph fibration*, a weakening of the notion of covering used in Angluin’s paper that also subsumes the concept of similarity of processors introduced in [JS85]. Moreover, the characterizations are stated in a completely uniform way across the activation models—the only change is in the family of fibrations considered.

We remark that the synchronous model is computationally equivalent to asynchronous FIFO networks with finite time delivery (which were the original reason behind the study of the model [YK96]).

The motivation for the study of impossibility results in anonymous networks stems also from questions about self-stabilizing systems: as already noted in [SRR95], and exploited in [BV97b], an impossibility result for anonymous networks gives an impossibility result for a uniform self-stabilizing system (as one of the choices of the adversary is setting all processors to the same state).

We begin by introducing a simple finite class of networks that is used to exemplify our (fairly abstract) characterization theorems. Then, we give our main definitions, recall the standard notion of view and introduce the main mathematical ingredient of our proofs—graph fibrations. Finally, we prove our characterization theorem and give some applications. For other examples of applications, the reader should be able to apply easily our characterization theorems to classical problems such as election, topology reconstruction or function computation, getting back the results of our previous papers.

## 2 A Guiding Example

As a guiding example, we formulate a problem that, to our knowledge, has not been previously addressed in the literature about anonymous networks. We are interested in determining whether there exist an anonymous algorithm solving the *majority problem* on a certain class of networks. All processors are given a boolean value (0 or 1) as input, and eventually must choose an output, the same for all processors, that corresponds to the majority of input values (in case of a tie, either value is correct, provided that all processors agree on that value). The algorithm must work on *every* network of the class under examination (informally speaking: processors just know that they live in one of the networks depicted, but they know neither which one exactly, nor which is their position

in the network). We shall use the class depicted in Figure 1 as a case study. For simplicity, in the example we restrict to the synchronous case.

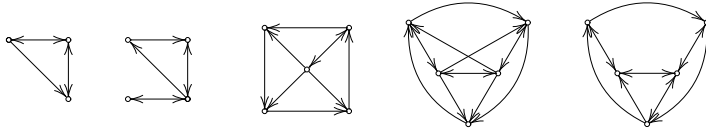


Fig. 1. Can you compute majority here?

### 3 Basic Definitions

#### 3.1 Graph-Theoretical Definitions

A (directed) (multi)graph  $G$  is given by a nonempty set  $N_G = \{1, 2, \dots, n_G\}$  of nodes and a set  $A_G$  of arcs, and by two functions  $s_G, t_G : A_G \rightarrow N_G$  that specify the source and the target of each arc. An (arc-)coloured graph (with set of colours  $C$ ) is a graph endowed with a colouring function  $\gamma : A_G \rightarrow C$ . We write  $i \xrightarrow{a} j$  when the arc  $a$  has source  $i$  and target  $j$ , and  $i \rightarrow j$  when  $i \xrightarrow{a} j$  for some  $a \in A_G$ . Subscripts will be dropped whenever no confusion is possible.

A (in-directed) tree is a graph with a selected node, the root, and such that any other node has exactly one directed path to the root.

#### 3.2 The Model

In the following, by a *network* we shall always mean a strongly connected coloured graph. The nodes of such a graph are called *processors*.

Computations of a network are defined by a state space and a transition function specifying how a processor must change its state when it is activated. The new state must depend, of course, on the previous state, and on the states of the in-neighbours; the latter are marked with the colour of the corresponding incoming arc (thus, e.g., if all incoming arcs have different colours a processor can distinguish its in-neighbours). For sake of simplicity, though, we discuss the case with just one colour, so that arcs and nodes are in fact not coloured; the introduction of more colours makes no significant conceptual difference.

We also need a function mapping input values to initial states and final states to output values. Formally, a *protocol*  $P$  is given by a set  $X$  of local states, a distinguished subset  $F \subseteq X$  of final states, an input set  $\mathcal{T}$ , an input function  $\text{in} : \mathcal{T} \rightarrow X$ , an output

<sup>1</sup> Since we need to manage infinite trees too, we assume that the node set of a tree can be  $\mathbb{N}$ .

<sup>2</sup> Colours on the nodes act as identifiers, whereas colours on the arcs define the communication model. Choosing a suitable colouring we can encode properties such as the presence of unique identifiers, distinguished outgoing/incoming links and so on [BCG<sup>+</sup>96].

set  $\Omega$ , an output function  $\text{out} : F \rightarrow \Omega$  and a transition function  $\delta : X \times X^\oplus \rightarrow X$  satisfying the constraint that for every  $x \in F$ ,  $\delta(x, -) = x$ ; here  $X^\oplus$  is the set of finite multisets over  $X$ .

Intuitively, the new state of a processor depends on its previous state (the first component of the cartesian product) and on the states of its in-neighbours (we must use multisets, as more than one in-neighbour might be in the same state). The additional condition simply states that when a processor enters a final state, it cannot change its own state thereafter, whatever input it receives from its in-neighbours.

A *global state* (for  $n$  processors, with respect to the protocol  $P$ ) is a vector  $\mathbf{x} \in X^n$ . Such a state is *final* if  $x_i \in F$  for every  $i$ .

Given a network  $G$  and a global state  $\mathbf{x}$ , we say that processor  $i$  is *enabled in  $\mathbf{x}$*  if an application of the transition function would change its state. Of course, no processor is enabled in a final state.

An *activation* for  $G$  in state  $\mathbf{x}$  is a nonempty set of enabled processors; the activation is called

- *synchronous* if it contains every enabled processor;
- *interleaved* if it contains but a single enabled processor.

When we need to emphasize that no constraint is required on the set of activated processors, we use the term *asynchronous*.

A *computation* of the protocol  $P$  on the network  $G$  (with  $n$  processors) with input  $\mathbf{v} \in \mathcal{Y}^n$  is given by a (possibly infinite) sequence of global states  $\mathbf{x}^0, \mathbf{x}^1, \dots, \mathbf{x}^T, \dots$  and a sequence of sets of processors  $A^0, A^1, \dots, A^{T-1}, \dots$  (called the *activation sequence*) such that:

1.  $\mathbf{x}^0 = \text{in}(\mathbf{v})$ ;
2. for all  $t$ ,  $A^t$  is an activation for  $G$  in state  $\mathbf{x}^t$ , and  $\mathbf{x}^{t+1}$  is obtained applying  $\delta$  to all processors of  $A^t$  (the other processors do not change their state).

A computation is called *synchronous*, *interleaved* or *asynchronous* if every activation is such. It is *terminating* if it is finite and its last state is final; in this case, its output is defined as the vector  $\text{out}(\mathbf{x}^T)$ .

Given a class of networks  $\mathcal{C}$ , a ( $\mathcal{C}$ -indexed) *relation*  $R$  is a function associating to each  $G \in \mathcal{C}$  a set  $R_G \subseteq \mathcal{Y}^n \times \Omega^n$ , where  $n$  is the number of processors of  $G$ . Relations are used to define problems: for instance, if  $\mathcal{Y} = \{*\}$  and  $\Omega = \{0, 1\}$ , the relation containing all pairs  $\langle * * \dots *, b_1 b_2 \dots b_n \rangle$ , where exactly one of the  $b_i$ 's is nonzero, defines the well-known *anonymous election problem*. Function computations, topology reconstruction etc. can be easily described in a similar way. Note that classes specify knowledge: the larger the class, the smaller the knowledge (the class of all networks corresponds to no knowledge at all; a singleton to maximum knowledge).

Let  $\mathcal{C}$  be a class of networks,  $R$  be a relation and  $P$  a protocol. We say that  $P$  *computes* (in a synchronous, interleaved, asynchronous manner resp.)  $R$  on  $\mathcal{C}$  if for all  $G \in \mathcal{C}$  with  $n$  processors it happens that for all  $\mathbf{v} \in \text{dom}(R_G)$ , every maximal (synchronous, interleaved, asynchronous resp.) computation of  $P$  on  $G$  with input  $\mathbf{v}$  is terminating, and its output  $\omega$  is such that  $\mathbf{v} R_G \omega$ .

<sup>3</sup> The input and output functions will be silently extended componentwise to vectors.

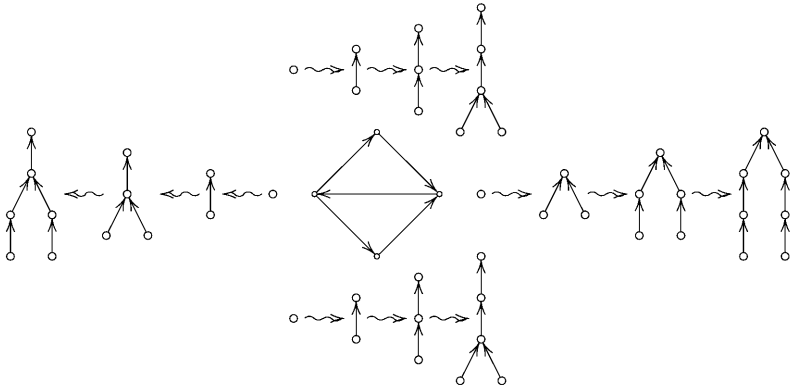
### 3.3 The View Construction

A classical tool in the study of anonymous networks is the concept of *view*, introduced for bidirectional networks in [YK96] and extended to the directed case in [BCG<sup>+</sup>96] (to be true, the concept of view is already present in the seminal paper [Ang80], partially disguised under the mathematical notion of *graph covering*). The view of a processor is a tree that gathers all the topological information that the processor can obtain by exchanging information with its neighbours.

More formally, the *view* of a processor  $i$  in the network  $G$  is an in-tree  $\tilde{G}^i$  built as follows:

- the nodes of  $\tilde{G}^i$  are the (finite) paths of  $G$  ending in  $i$ , the root of  $\tilde{G}^i$  being the empty path;
- there is an arc from the node  $\pi$  to the node  $\pi'$  if  $\pi$  is obtained by adding an arc  $a$  at the beginning of  $\pi'$ .

The tree  $\tilde{G}^i$  is always infinite if  $G$  is strongly connected and has at least one arc, and there is a trivial anonymous protocol that allows each processor to compute its own view truncated at any desired depth. At step  $k + 1$  of the protocol, each processor gathers from its in-neighbours their views truncated at depth  $k$ , and combining them it can compute its own view truncated at depth  $k + 1$ . The start state is the one-node tree. An example of view construction is given in Figure 2, where we show the first four steps of view construction for the processors of a simple network.



**Fig. 2.** An example of view construction

Clearly, in the view construction process, inputs must be taken into account. That is, the views are really coloured on their nodes by elements on the input set. It is possible that two processor have the same view with a certain choice of inputs, but have a different view with another. For instance, if all processors have different inputs, then all processors get different views, even in the case of a very symmetric topology (e.g., a ring).

The reason why views are important is that the state of a processor after  $k$  steps of any anonymous computation may only depend on its view truncated at depth  $k$ . Thus, it is of crucial importance to determine which processors of a network possess the same (infinite) view.

### 3.4 Graph Fibrations and the Lifting Lemma

The problems stated at the end of the last section can be solved very elegantly using an elementary graph-theoretical concept, that of *graph fibration* [BV]. A fibration formalizes the idea that processors that are connected to processors behaving in the same way will behave alike; it generalizes both the usage of graph coverings in Angluin's original paper [Ang80] and the concept of similarity of processors introduced in [US85].

Recall that a *graph morphism*  $f : G \rightarrow H$  is given by a pair of functions  $f_N : N_G \rightarrow N_H$  and  $f_A : A_G \rightarrow A_H$  that commute with the source and target functions, that is,  $s_H \circ f_A = f_N \circ s_G$  and  $t_H \circ f_A = f_N \circ t_G$ . (The subscripts will usually be dropped.) In other words, a morphism maps nodes to nodes and arcs to arcs in such a way to preserve the incidence relation. If colours are present (on the arcs or on the nodes) they must be preserved.

**Definition 1** A fibration between (coloured) graphs  $G$  and  $B$  is a morphism  $\varphi : G \rightarrow B$  such that for each arc  $a \in A_B$  and for each node  $i \in N_G$  satisfying  $\varphi(i) = t(a)$  there is a unique arc  $\tilde{a}^i \in A_G$  (called the *lifting* of  $a$  at  $i$ ) such that  $\varphi(\tilde{a}^i) = a$  and  $t(\tilde{a}^i) = i$ .

If  $\varphi : G \rightarrow B$  is a fibration,  $B$  is called the *base* of the fibration. We shall also say that  $G$  is *fibred* (over  $B$ ). The *fibre* over a node  $i \in N_B$  is the set of nodes of  $G$  that are mapped to  $i$ , and will be denoted by  $\varphi^{-1}(i)$ .

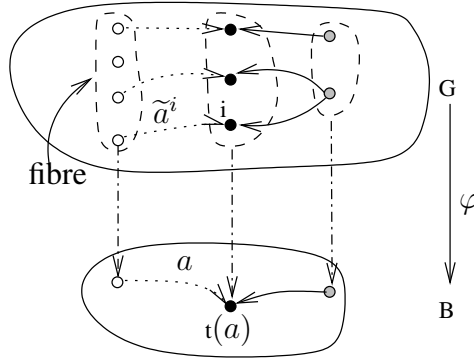
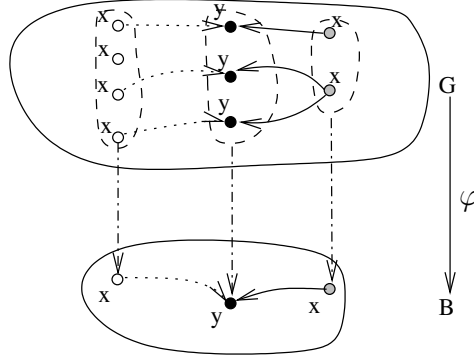
There is a very intuitive characterization of fibrations based on the concept of local isomorphism. A fibration  $\varphi : G \rightarrow B$  induces an equivalence relation between the nodes of  $G$ , whose classes are precisely the fibres of  $\varphi$ . When two nodes  $i$  and  $j$  are equivalent (i.e., they are in the same fibre), there is a bijective correspondence between arcs coming into  $i$  and arcs coming into  $j$  such that the sources of any two related arcs are equivalent.

In Figure 3 we sketched a fibration between two graphs. Note that, because of the lifting property described in Definition 1, all black nodes have exactly two incoming arcs, one (the dotted arc) going out of a white node, and one (the continuous arc) going out of a grey node. In other words, the in-neighbour structure of all black nodes is the same.

The main *raison d'être* of fibrations is that they allow to relate the behaviour of the same protocol on two networks. To make this claim precise, we need some notation: if  $\varphi : G \rightarrow B$  is a fibration, and  $\mathbf{x}$  is a global state of  $B$ , we can obtain a global state  $\mathbf{x}^\varphi$  of  $G$  by “lifting” the global state of  $B$  along each fibre, that is,  $(\mathbf{x}^\varphi)_i = x_{\varphi(i)}$  (see Figure 4). Essentially, starting from a global state of  $B$  we obtain a global state of  $G$  by copying the state of a processor of  $B$  fibrewise.

**Lemma 1 (Lifting Lemma [BV97a]).** Let  $\varphi : G \rightarrow B$  be a fibration. Then, for every protocol  $P$  and every synchronous computation  $\mathbf{x}^0, \mathbf{x}^1, \dots$  of  $P$  on  $B$ ,  $(\mathbf{x}^0)^\varphi, (\mathbf{x}^1)^\varphi, \dots$  is a synchronous computation of  $P$  on  $G$ .



**Fig. 3.** A fibration.**Fig. 4.** State lifting.

The above lemma suggests that if a network can be “collapsed” by a fibration onto another one, then there are certain constraints on its behaviours. More precisely, we can compute the sequence of lifted global states of  $G$  by computing the sequence of global states of  $B$ , and then lifting the result. In other words, the computation of  $B$  “summerizes” the computation of  $G$ .

In particular, the computation above terminates in  $T$  steps on  $B$  iff the lifted computation does the same on  $G$ . Moreover, the outputs of the two computations are clearly related: the output of a processor of  $G$  is exactly the output of the processor of  $B$  it is mapped to by the fibration.

Note that the above lemma does not provide constraints on *all* computations of  $G$ . Rather, it works when inputs are assigned fibrewise (or, from a symmetrical point of view, when they are lifted from  $B$ ). In this case, it tells us that we can really do the computation on  $B$  and lift the result at the end.

These considerations may seem trivial when applied to a single network, but things become trickier when a whole class is involved. Indeed, if *two* different networks  $G$  and  $H$  of the class are fibred over a common base  $B$ , then the output of the two networks with respect to inputs that are liftings of the same input of  $B$  are inextricably related.

Indeed, these observations are already sufficient to show that *majority is not computable in the class of Figure 1*. In Figure 5 we show that two networks  $G$  and  $H$  of our class are fibred over a common base  $B$ . Suppose by contradiction that you have a protocol computing the majority. If we run this protocol on  $B$  with, say, input 0 to  $a$  and 1 to  $b$ , it will terminate with output  $\omega$  (0 or 1 are both correct; note that it must terminate, because it terminates on  $G$  and  $H$ ). However, if we lift this input to  $G$  and  $H$ ,  $\omega$  will not be a correct output for one of them—contradiction.

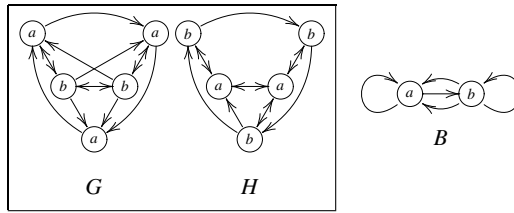


Fig. 5. Two networks with a common base.

## 4 Characterizing Solvability for Arbitrary Classes

Using fibration we have proved an impossibility result for our example. However, there is a missing link: processor can build views, but are constrained in their behaviour by the bases of the network they live in. The connection between these two concepts lies in the notion of *minimum base*.

### 4.1 Minimum Bases

We say that a graph  $G$  is *fibration prime* if every fibration from  $G$  is an isomorphism, that is,  $G$  cannot be collapsed onto a smaller network by a fibration.

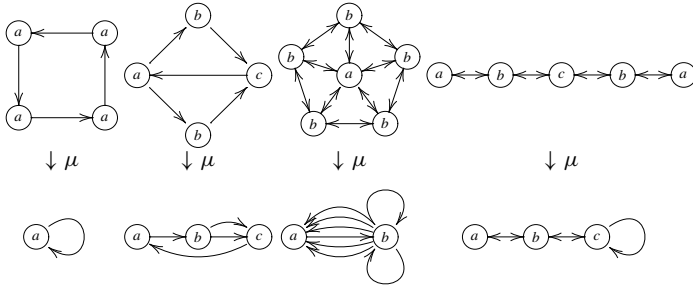
**Theorem 1 ([BV])** *The following properties hold:*

1. For every graph  $G$  there is (up to isomorphism) exactly one fibration-prime graph  $\hat{G}$  such that  $G$  is fibred onto (i.e., surjectively)  $\hat{G}$ .
2. If  $G$  and  $H$  have a common base  $B$ , then  $\hat{G} \cong \hat{H}$ .
3. Let  $\varphi : G \rightarrow \hat{G}$ ; then  $\tilde{G}^i \cong \tilde{G}^j$  iff  $\varphi(i) = \varphi(j)$  (as a consequence, distinct nodes of a fibration prime graph have different views).
4. A fibration-prime graph is uniquely characterized by the set of views of its nodes.

There are many ways to build  $\hat{G}$ . One is a *partition set* algorithm [BV], similar to the one used for finite-state automaton minimization. On the other hand, one can also take as nodes of  $\hat{G}$  the distinct views of  $G$ , and put an arc between two views if one view is a first-level child of the second one.

The fibrations from  $G$  to  $\hat{G}$  are called *minimal*. There is usually more than one minimal fibration, but they must all coincide on the nodes by property (3) of Theorem 1, so we denote with  $\mu_G$  one of them when the map on the arcs is not relevant.

In Figure 6 we show a number of networks and the corresponding minimum bases. Again, the node component of a minimal fibration is represented by suitably labelling the nodes. Another example of minimum base was given in Figure 5 the graph  $B$  was the minimum base of both  $G$  and  $H$ , that is,  $\hat{G} \cong \hat{H} \cong B$ .



**Fig. 6.** Some examples of minimal fibrations and minimum bases.

The previous theorem highlights the deep link between fibrations and views: two processors have the same view if and only if they lie in the same fibre of  $\mu$ . Since we know by the Lifting Lemma that processors in the same fibre of a fibration cannot behave differently, this means that on the one hand, processors with the same view will always be in the same state, and on the other hand, if we can deduce  $\hat{G}$  from a view we can use the Lifting Lemma to characterize the possible behaviours.

The fundamental fact we shall use intensively in all proofs is that the above considerations, which involve infinite objects (isomorphism of infinite trees, and so on), can be described by means of finite entities using the following theorem:

**Theorem 2 ([BV])** *Let  $G$  be a strongly connected graph and  $B$  a fibration-prime graph with minimum number of nodes such that, for some node  $i$  of  $G$  and node  $j$  of  $B$ ,  $\hat{G}^i$  and  $\hat{B}^j$  are isomorphic up to height  $n_G + (\text{diameter of } G)$ : then  $B \cong \hat{G}$ , and  $i$  is mapped to  $j$  by all minimal fibrations.*

How can a processor use the (apparently unfathomable) previous theorem to build the minimum base? First of all, in  $2N - 1$  rounds all processors build their views up to height  $2N - 1$ , where  $N$  is a known bound on the number of processors (and thus

$N - 1$  bounds the diameter). Then, they perform locally an exhaustive search for the only graph  $B$  satisfying the theorem.<sup>4</sup>

Note that since our processors have additional information given by their inputs, the constructions described here must be performed *taking the inputs into account*. More precisely, views and minimum bases must be constructed “as if” the processors were (additionally) coloured with their inputs. For instance, in Figure 7 we show the minimum base for a 4-cycle with respect to different inputs.

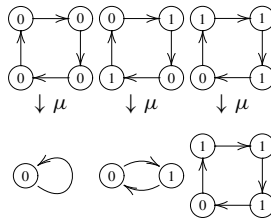


Fig. 7. Minimal fibrations of the same graph with respect to different inputs.

## 4.2 The (A)Synchronous Case

We are finally ready to characterize the classes of networks on which (a)synchronous computation of a relation  $R$  is possible. First of all we notice that, from a computability viewpoint, there is no difference between the synchronous and the asynchronous case, as shown in [BV99]; thus, we restrict our proofs to the synchronous case.

**Theorem 3** *Let  $\mathcal{C}$  be a class of networks of bounded size<sup>5</sup>. Then  $R$  is (a)synchronously computable on  $\mathcal{C}$  iff for all graphs  $B$  and all  $\mathbf{v} \in \Upsilon^{n_B}$  there is an  $\omega \in \Omega^{n_B}$  such that for all  $G \in \mathcal{C}$  and all fibrations  $\varphi : G \rightarrow B$ , if  $(\mathbf{v})^\varphi \in \text{dom}(R_G)$  then  $(\mathbf{v})^\varphi R_G (\omega)^\varphi$ .*

*Proof.* If the conditions of the statement are not satisfied, then there is a graph  $B$  and a  $\mathbf{v} \in \Upsilon^{n_B}$  such that for all  $\omega \in \Omega^{n_B}$  there is a  $G \in \mathcal{C}$  and a fibration  $\varphi : G \rightarrow B$  such that  $(\mathbf{v})^\varphi \in \text{dom}(R_G)$  but not  $(\mathbf{v})^\varphi R_G (\omega)^\varphi$ . By the Lifting Lemma this implies that, whichever protocol we use, at least one computation on some graph  $G$  in  $\mathcal{C}$  starting from  $\text{in}((\mathbf{v})^\varphi)$  will give an output that is incorrect.

Otherwise, each processor determines its minimum base taking into account the inputs, and, using the conditions above, chooses an output value that satisfies  $R_G$  (i.e., it looks for the correct output on the minimum base and chooses its own output accordingly).

<sup>4</sup> There are of course “smarter” ways to find  $\hat{G}$  for specific types of networks; nonetheless, for the general case Theorem 2 gives an optimal bound [BV].

<sup>5</sup> By this we mean that there is an integer  $N$  such that all networks in  $\mathcal{C}$  have at most  $N$  nodes.

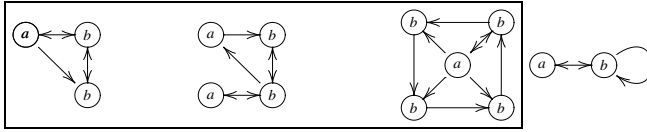
The characterization implied by statement of the previous theorem may seem to be daunting, but it really boils down to the following simple procedure:

1. Compute the minimum bases of the networks in  $\mathcal{C}$  with respect to every possible input assignment.
2. For each base, choose an output  $\omega$  satisfying the conditions of the theorem.
3. If the previous step is not possible, then the relation cannot be computed on  $\mathcal{C}$ ; otherwise, the chosen outputs can be put into a table and used by the processors, after the minimum base construction, to choose their output.

All the above steps are finite if  $\mathcal{C}$  and  $R$  are finite. Thus, the characterization is recursive.

We can now apply the characterization to our original problem. First of all, we have to throw away one of the last two networks: as we already argued (and as easily shown using the above theorem) there is no majority algorithm for the class if both networks of Figure 5 are present. Hence, we restrict to the first four networks of Figure 1.

We note that the first three networks in Figure 1 have a different minimum base than the others, as shown in Figure 8; thus, we can consider them separately.



**Fig. 8.** The remaining networks and their minimum base.

Now, a trivial application of the procedure we outlined shows that the conditions of Theorem 3 are satisfied: the only subtle point is that if we assign different inputs to the two nodes of the minimum base, there is always a choice for the output that works when lifted to the three networks.

A similar investigation shows, for instance, that election is not solvable in the restricted class of Figure 8: no matter which processor we decide to elect on the base, there will be two processors elected in the second network. If the latter is eliminated, however, election becomes possible.

### 4.3 The Interleaved Case

We sketch the ideas behind the characterization of the interleaved case. We say that a fibration is *acyclic* if the subgraphs induced by each fibre are acyclic (except for loops). Acyclicity guarantees that there is always an interleaved activation working fibrewise, so that the Lifting Lemma can be extended to this case (as done in [BV97a]). More precisely, for each interleaved activation on the base  $B$  of a fibration  $\varphi : G \rightarrow B$  one can find an interleaved activation of  $G$  such that processors are activated fibre by fibre. The activation in each fibre starts from a sink in the fibre and continues inductively so that no processor in the fibre is activated before one of its successors in the fibre.

On the positive side, processors use a simple 0–1 game (as shown in [BCG<sup>+</sup>96] for election) to break asymmetry until they obtain a minimal *acyclic* fibration (that is, a fibration that collapses a network as much as possibly without violating acyclicity). The theory in this case is a bit different, as there are many possible, nonisomorphic, minimal bases. Nonetheless, the statement of the characterization remains the same:

**Theorem 4** *Let  $\mathcal{C}$  be a class of networks of bounded size. Then  $R$  is computable with interleaved activation on  $\mathcal{C}$  iff for all graphs  $B$  and all  $v \in \mathcal{T}^{n_B}$  there is an  $\omega \in \Omega^{n_B}$  such that for all  $G \in \mathcal{C}$  and all acyclic fibrations  $\varphi : G \rightarrow B$ , if  $(v)^\varphi \in \text{dom}(R_G)$  then  $(v)^\varphi R_G (\omega)^\varphi$ .*

*Proof.* If the conditions of the statement are not satisfied, then there is a graph  $B$  and a  $v \in \mathcal{T}^{n_B}$  such that for all  $\omega \in \Omega^{n_B}$  there is a  $G \in \mathcal{C}$  and an acyclic fibration  $\varphi : G \rightarrow B$  such that  $(v)^\varphi \in \text{dom}(R_G)$  but not  $(v)^\varphi R_G (\omega)^\varphi$ . By the Lifting Lemma (extended to acyclic fibrations and interleaved activations) this implies that, whichever protocol we use, at least one computation on some graph  $G$  starting from  $\text{in}((v)^\varphi)$  will give an output that is incorrect.

With respect to the proof of Theorem 3, we have to show that we can break symmetry in such a way to obtain a fibration  $\varphi : G \rightarrow B$  which is acyclic. To this purpose, the processors first compute a standard minimal fibration (taking inputs into account); then, processors in the same fibre label themselves either 0 or 1 according to the following rule:

- if the processor is activated and all in-neighbours in its fibre are unlabelled, it labels itself 0;
- if the processor is activated and any in-neighbours in its fibre is labelled, it labels itself 1.

At the end of this “0–1 game”, the processors compute again the fibres of a minimal fibration, this time taking into account the 0–1 identifier they possess. This process can be repeated, and can only terminate when the minimal fibration is acyclic; this happens because at each step every nontrivial fibre containing an oriented cycle breaks at least into two pieces (in such a fibre, the first activated processor labels itself 0, and among the processors belonging to an oriented cycle the least activated processor necessarily labels itself 1). Now all processors can select an output using the conditions in the statement.

If we get back to our guiding example, we can note that all networks in Figure 1 are minimal with respect to acyclic fibrations, except for the third, which is acyclically fibred over the first one. Again, a simple case-by-case examination shows that now both majority and election are possible, due to the great desymmetrizing power of interleaved activation.

## 5 Some Easy Applications

We give a few theorems outlining more general applications of our results. To our knowledge, these are the first results for these problems (the results apply also to the interleaved model).

**Theorem 5** *Let  $\mathcal{C}$  be the class of all bidirectional networks of size  $n$  (that is, every arc has an associated arc in the opposite direction). Then majority is computable on  $\mathcal{C}$ .*

*Proof.* (Sketch) The linear constraints imposed by bidirectionality determine uniquely the cardinality of the fibres of a fibration once its base is known [BV]. Thus, the output vector  $\omega$  in the statement of Theorem 3 can be chosen by counting the actual multiplicity of each input.

We can give a more elementary point of view on the previous theorem: by solving a linear system, each processor can determine the cardinality of all fibres of the minimal fibration. Then, it can compute locally the multiplicity of each input, and thus derive the correct output.

Note that the above theorem is independent of the colouring on the arcs, and thus of the communication model. For instance, it works in broadcast networks without distinguished incoming links.

**Theorem 6** *Let  $\mathcal{C}$  be a class of networks with distinguished outgoing links (that is, every arc has a colour and arcs going out of the same processor get different colours) and bounded size. Then majority is computable on  $\mathcal{C}$ .*

*Proof.* Since the outgoing links have different colours, all fibrations starting from a network in  $\mathcal{C}$  have the property that all fibres have the same cardinality (this happens because they are really *coverings*—see [BV]). Thus, lifting multiplies each input the same number of times; hence, the output vector  $\omega$  can be chosen by taking a majority vote on  $v$ .

The theorem above would not hold if we required distinguished *incoming* links. A counterexample can be built by colouring all arcs of the networks  $B$  of Figure 5 with different colours, and lifting the colours on the networks  $G$  and  $H$ .

It is interesting to contrast the behaviour of the *counting* problem in the above two cases: each processor must compute how many processor with its own input are present.

**Theorem 7** *Let  $\mathcal{C}$  be the class of all bidirectional networks of size  $n$  (that is, every arc has an associated arc in the opposite direction). Then counting is computable on  $\mathcal{C}$ .*

**Theorem 8** *Let  $\mathcal{C}$  be a class of networks with distinguished outgoing links (that is, every arc has a colour and arcs going out of the same processor get different colours) and size  $n$ . Then counting is computable on  $\mathcal{C}$ .*

The theorem above does *not* hold if just a bound on the size is known, and, of course, if we required distinguished incoming links. The proof of the last statements should be now an easy exercise for the reader.

## 5.1 Conclusions

We have only surfaced the range of applications of the general theory of anonymous computation sketched in this paper. For instance, the theory also allows one to establish the possibility of *weak* computability. A relation is *weakly* computable on a class  $\mathcal{C}$  if there is a protocol that works correctly on all networks of the class on which the relation is computable, but stops all processors in a special “impossibility state” in the remaining networks. In other words, the processors either compute the result, or establish in a distributed and anonymous way that this is impossible on specific network they live in (previous literature often confused the strong and weak version of a problem). Of course, in general the classes on which a relation is weakly computable are much larger than in the standard case. It is not difficult to derive from Theorem 3 a necessary and sufficient condition for weak computability.

As we noticed elsewhere, the bound given on the number of steps required to compute a relation (the number of nodes plus the diameter) is optimal, as there are classes of graphs in which topology reconstruction cannot be performed in less than that number of steps.

## References

- [Ang80] Dana Angluin. Local and global properties in networks of processors. In *Proc. 12th Symposium on the Theory of Computing*, pages 82–93, 1980.
- [ANIM96] Nechama Allenberg-Navony, Alon Itai, and Shlomo Moran. Average and randomized complexity of distributed problems. *SIAM Journal on Computing*, 25(6):1254–1267, 1996.
- [ASW88] Hagit Attiya, Marc Snir, and Manfred K. Warmuth. Computing on an anonymous ring. *J. Assoc. Comput. Mach.*, 35(4):845–875, 1988.
- [BCG<sup>+</sup>96] Paolo Boldi, Bruno Codenotti, Peter Gemmell, Shella Shammah, Janos Simon, and Sebastiano Vigna. Symmetry breaking in anonymous networks: Characterizations. In *Proc. 4th Israeli Symposium on Theory of Computing and Systems*, pages 16–26. IEEE Press, 1996.
- [BV] Paolo Boldi and Sebastiano Vigna. Fibrations of graphs. *Discrete Math.* To appear.
- [BV97a] Paolo Boldi and Sebastiano Vigna. Computing vector functions on anonymous networks. In Danny Krizanc and Peter Widmayer, editors, *SIROCCO '97. Proc. 4th International Colloquium on Structural Information and Communication Complexity*, volume 1 of *Proceedings in Informatics*, pages 201–214. Carleton Scientific, 1997. An extended abstract appeared also as a Brief Announcement in *Proc. PODC '97*, ACM Press.
- [BV97b] Paolo Boldi and Sebastiano Vigna. Self-stabilizing universal algorithms. In Sukumar Ghosh and Ted Herman, editors, *Self-Stabilizing Systems (Proc. of the 3rd Workshop on Self-Stabilizing Systems, Santa Barbara, California, 1997)*, volume 7 of *International Informatics Series*, pages 141–156. Carleton University Press, 1997.
- [BV99] Paolo Boldi and Sebastiano Vigna. Computing anonymously with arbitrary knowledge. In *Proc. 18th ACM Symposium on Principles of Distributed Computing*, pages 181–188. ACM Press, 1999.
- [DKMP95] Krzysztof Diks, Evangelos Kranakis, Adam Malinowski, and Andrzej Pelc. Anonymous wireless rings. *Theoret. Comput. Sci.*, 145:95–109, 1995.
- [JS85] Ralph E. Johnson and Fred B. Schneider. Symmetry and similarity in distributed systems. In *Proc. 4th conference on Principles of Distributed Computing*, pages 13–22, 1985.



- [Nor96] Nancy Norris. Computing functions on partially wireless networks. In Lefteris M. Kirousis and Evangelos Kranakis, editors, *Structure, Information and Communication Complexity. Proc. 2nd Colloquium SIROCCO '95*, volume 2 of *International Informatics Series*, pages 53–64. Carleton University Press, 1996.
- [NS95] Moni Naor and Larry Stockmeyer. What can be computed locally? *SIAM J. Comput.*, 24(6):1259–1277, 1995.
- [SRR95] Sandeep K. Shukla, Daniel J. Rosenkrantz, and S.S. Ravi. Observations on self-stabilizing graph algorithms for anonymous networks. In *Proceedings of the Second Workshop on Self-Stabilizing Systems*, pages 7.1–7.15, Las Vegas, 1995. University of Nevada.
- [YK96] Masafumi Yamashita and Tiko Kameda. Computing on anonymous networks: Part I—characterizing the solvable cases. *IEEE Trans. Parallel and Distributed Systems*, 7(1):69–89, 1996.
- [YK98] Masafumi Yamashita and Tiko Kameda. Erratum to computing functions on asynchronous anonymous networks. *Theory of Computing Systems (formerly Math. Systems Theory)*, 31(1), 1998.

# Competitive Hill-Climbing Strategies for Replica Placement in a Distributed File System

John R. Douceur and Roger P. Wattenhofer

Microsoft Research, Redmond WA 98052, USA  
{johndo, rogerwa}@microsoft.com  
<http://research.microsoft.com>

**Abstract.** The Farsite distributed file system stores multiple replicas of files on multiple machines, to provide file access even when some machines are unavailable. Farsite assigns file replicas to machines so as to maximally exploit the different degrees of availability of different machines, given an allowable replication factor  $R$ . We use competitive analysis and simulation to study the performance of three candidate hill-climbing replica placement strategies, **MinMax**, **MinRand**, and **RandRand**, each of which successively exchanges the locations of two file replicas. We show that the **MinRand** and **RandRand** strategies are perfectly competitive for  $R = 2$  and  $2/3$ -competitive for  $R = 3$ . For general  $R$ , **MinRand** is at least  $1/2$ -competitive and **RandRand** is at least  $10/17$ -competitive. The **MinMax** strategy is not competitive. Simulation results show better performance than the theoretic worst-case bounds.

## 1 Introduction

This paper analyzes algorithms for automated placement of file replicas in the Farsite [3] system, using both theory and simulation. In the Farsite distributed file system, multiple replicas of files are stored on multiple machines, so that files can be accessed even if some of the machines are down or inaccessible. The purpose of the placement algorithm is to determine an assignment of file replicas to machines that maximally exploits the availability provided by machines.

The file placement algorithm is given a fixed value,  $R$ , for the number of replicas of each file. For systems reasons, we are most interested in a value of  $R = 3$  [9]. However, to ensure that our results are not excessively sensitive to the file replication factor, we also provide tight bounds for  $R = 2$  and lower bounds for all  $R$  (tight at different values of  $R$ ).

Our theoretic investigations cover an arbitrary distribution of machine availabilities and show worst-case behavior for a slightly abstracted model of the problem. For these studies, we assume an adversary that can establish – and continuously change – the availability characteristics for all machines, and we assess the ability of our algorithms to maximize the minimum file availability relative to the optimally achievable minimum file availability, for any given assignment of machine availability values. We do not attempt to classify the computational complexity of the problem, because it is not a classic input-output algorithm.

Our simulations are driven by actual measurements of machine availability [9] and show average-case behavior for a specific set of real-world measurements. For these studies, we consider not only the minimum file availability but also the distribution of file availability values. In all cases, we use a logarithmic measure for machine and file availability values, in part because of its standard usage [15] and computational convenience, but also because a linear measure understates the differences between results, since for all algorithms the minimum-availability file is available for a fraction of time that is very close to unity.

The next section describes the Farsite system and provides some motivation for why file replica placement is an important problem. Section 3 describes the algorithms, followed by a summary of results in Section 4. Section 5 presents a simplified theoretic model of the Farsite system environment, which is used in Section 6 to analyze the performance of the algorithms. Section 7 describes the environment for our simulations, the results of which are detailed in Section 8. Related work is discussed in Section 9.

## 2 Background

Farsite [3] is a secure, highly scalable, serverless, distributed file system that logically functions as a centralized file server without requiring any physical centralization whatsoever. The system's computation, communication, and storage are distributed among all of the client computers that participate in the system. Farsite runs on a networked collection of ordinary desktop computers in a large corporation or university, without interfering with users' local tasks, and without requiring users to modify their behavior in any way. As such, it needs to provide a high degree of security and fault tolerance without benefit of the physical protection and continuous support enjoyed by centralized server machines.

There are four properties that Farsite provides for the files that it stores: privacy, integrity, reliability, and availability. Data privacy is afforded by symmetric-key and public-key encryption, and data integrity is afforded by one-way hash functions and digital signatures. Reliability, in the sense of data persistence, is provided by making multiple replicas of each file and storing the replicas on different machines. The topic of the present paper is file availability, in the sense of a user's being able to access a file at the time it is requested.

Like reliability, file availability is provided by storing multiple replicas of each file on different machines. However, whereas the probability of permanent data-loss failure is assumed to be identical for all machines, the probability of transitory unavailability (such as a machine's being powered off temporarily) is demonstrably not identical for all machines. A five-week series of hourly measurements of more than 50,000 desktop machines at Microsoft [9] has shown that (1) machine availabilities vary dramatically from machine to machine, (2) the measured availability of each machine is reasonably consistent from week to week, and (3) the times at which different machines are unavailable are not significantly correlated with each other.

A file is not available if all the machines that store the replicas of the file are temporarily down. Given uncorrelated machine downtimes, the fraction of time that a file is unavailable is equal to the product of the fractional downtimes of the machines that store replicas of that file. We express availability as the negative logarithm of fractional downtime, and then the availability of a file is equal to the sum of the availabilities of the machines that store the file’s replicas. The goal of a file placement algorithm is to produce an assignment of file replicas to machines that maximizes the minimum file availability over all files without exceeding the available space on any machine.

Measurements of over 10,000 file systems on desktop computers at Microsoft [3] indicate that machines experience permanent data-loss failures (e.g. disk head crashes) in a temporally uncorrelated fashion. We do not allow the algorithm to vary the number of replicas on a per-file basis, because this would introduce variance into the distribution of file reliability. The measurements show that a value of  $R = 3$  is achievable in a real-world setting [9].

### 3 Algorithms

To be suitable for a distributed file system, a replica placement algorithm must satisfy two essential requirements: It must be incremental and distributable. Because the system environment is constantly changing, the algorithm must be able to improve an existing placement iteratively, rather than requiring a complete re-allocation of storage resources when a file is created or deleted, when a machine arrives or departs, or when a machine’s availability changes. Because the file system is distributed, the algorithm must scale with the size of the system and must operate by making small changes of strictly local scope. To satisfy these requirements, we concentrate on hill-climbing algorithms, in particular those that perform an ordered succession of swap operations, in which the machine locations of two file replicas are exchanged.

Specifically, we investigate the properties of three algorithms: (1) **MinMax**, in which the only allowed replica-location swaps are between the file with the minimum availability and the file with the maximum availability, (2) **MinRand**, in which swaps are allowed only between the file with the minimum availability and any other file, and (3) **RandRand**, in which swaps are allowed between any pair of files. In general, file replicas are swapped between machines only if the swap reduces the absolute difference between the availabilities of the files and only if there is sufficient free space on each machine to accept the replicas that are being relocated. If there is more than one successful swap for two given files, our algorithm chooses one with minimum absolute difference between the file availabilities after the swap. However, for our theoretical worst-case analysis, this does not matter.

The intuition behind these algorithms is as follows: **RandRand** is the most general swap-based strategy, in that it allows swaps between any pair of files, so it represents a baseline against which to compare and contrast the other algorithms. We are most concerned with improving the minimum file availability,

and since a replica exchange only affects the two files whose replica locations are swapped, it makes sense for one of these files to be the one with minimum availability, hence **MinRand**. The motivation behind **MinMax** is that the maximum availability file seems likely to afford the most opportunity for improving the minimum availability file without excessively decreasing its own availability.

In actual practice, the file placement algorithm executes in a distributed fashion, wherein the files are partitioned into disjoint sets, and each set is managed by an autonomous group of a few machines. At each iterative step, one of the groups contacts another group (possibly itself), each of the two groups selects one of the files it manages, and the groups jointly determine whether to exchange machine locations of one of the replicas of each file. Therefore, the **MinMax** and **MinRand** algorithms are not guaranteed to select files with globally extremal availability values. For our theoretic analyses, we concentrate on the more restrictive case in which only extremal files are selected. For our simulation studies, we model this extremal discrepancy by selecting from a range of files with the highest or lowest availability rank.

## 4 Summary of Results

In this paper we perform a worst-case analysis and a simulation to determine the efficacy of three hill-climbing algorithms, where the efficacy of an algorithm is specified by the availability of a file with minimum availability. We denote the efficacy of an algorithm by its competitive ratio  $\rho = m/m^*$ , where  $m$  is the efficacy of the hill-climbing algorithm, and  $m^*$  is the efficacy of an optimal algorithm. We show – for both theory and simulation – that the **MinRand** algorithm performs (almost) as well as the **RandRand** algorithm. The **MinMax** algorithm performs poorly throughout. Here is a detailed summary of our results:

Algorithm	MinMax	MinRand	RandRand
Worst-case $R = 3$	$\rho = 0$ (Thm. <a href="#">3</a> )	$\rho = 2/3$ (Thm. <a href="#">11</a> )	$\rho = 2/3$ (Thm. <a href="#">11</a> )
Simulated $R = 3$	$\rho \approx 0.74$ (Fig. <a href="#">2</a> )	$\rho \approx 0.93$ (Fig. <a href="#">2</a> )	$\rho \approx 0.91$ (Fig. <a href="#">2</a> )
Worst-case $R = 2$	$\rho = 0$ (Thm. <a href="#">3</a> )	$\rho = 1$ (Thm. <a href="#">2</a> )	$\rho = 1$ (Thm. <a href="#">2</a> )
Lower bounds any $R$	$\rho = 0$ (Thm. <a href="#">3</a> )	$\rho > 1/2$ (Thm. <a href="#">4</a> )	$\rho \geq 10/17$ (Thm. <a href="#">5</a> )

## 5 Theoretic Model

We are given a set of  $N$  unit-size files, each of which has  $R$  replicas. We are also given a set of  $M = N \cdot R$  machines, each of which has the capacity to store a single file replica. Machines have *availabilities*  $a_i \geq 0$ ,  $i = 1, \dots, M$ , given as negative logarithms of machines' downtimes.

Let the  $R$  replicas of file  $f$  be stored on machines with availabilities  $a_1^f, \dots, a_R^f$ . To avoid notational clutter, we overload a variable to name a file and to give the availability value of the file. Thus, the *availability* of file  $f$  is  $f = a_1^f + \dots + a_R^f$ .

Let  $m$  be a file with minimum availability when the algorithm has exhausted all possible improvements. Let  $m^*$  be a file with minimum availability given an optimal placement for the same values of  $N$ ,  $R$ , and  $a_i$  ( $i = 1, \dots, M$ ). We compute the ratio  $\rho = \min m/m^*$  over all allowable  $a_i$  as  $N \rightarrow \infty$ . We say that the algorithm is  $\rho$ -competitive.

In the practical algorithms, the particle “**Rand**” stands for a random choice, i.e. the **MinRand** algorithm tries to exchange machines between minimum-availability file  $m$  and a *randomly* chosen file  $f$ . In the theoretical analysis however, “**Rand**” is treated as “**Any**”, i.e. the **MinRand** algorithm tries to exchange machines between the minimum-availability file  $m$  and *any* other file  $f$ . The algorithm stops (“freezes”) only after all legal pairs of files have been tested. We use the particle “**Rand**” rather than “**Any**” to have a consistent terminology to the simulation part of this work.

If two or more files have minimum availability, or if two or more files have maximum availability, we allow an adversary to choose which of the files can be swapped.

The number of possible machine exchanges between two given files grows exponentially with the number of replicas  $R$ . In this paper we do not study this problem. We are predominantly interested in systems where  $R$  is small; for large  $R$  we will give an recipe linear in  $R$  that finds machines to be exchanged (see Lemma 4).

Note that the set of legal pairs of files for the **MinRand** algorithm is a subset of the set of legal pairs of files for the **RandRand** algorithm. That is, if the **MinRand** algorithm freezes, it is possible that the **RandRand** algorithm would still find a successful exchange. A freeze of the **RandRand** algorithm however also implies that the **MinRand** algorithm would not find a successful exchange. Similarly, the singleton set of legal pairs for **MinMax** is a subset of the legal pairs for **MinRand**. Thus, the efficacy of the **RandRand** (**MinRand**) algorithm is at least as high as the efficacy of the **MinRand** (**MinMax**) algorithm. Formally,

**Lemma 1.**  $\rho_{\text{MinMax}} \leq \rho_{\text{MinRand}} \leq \rho_{\text{RandRand}}$ .

With Lemma 1 we are in the position to find the competitive ratio of different algorithms by simply giving a worst-case example for the stronger algorithm (e.g. **RandRand**), and a qualitative argument on the weaker algorithm (e.g. **MinRand**). When discussing the competitive ratio of an algorithm with a specific replication factor, we append the replication factor  $R$  to the algorithm name, i.e. the competitive ratio of the **MinRand** algorithm for replication factor 3 is  $\rho_{\text{MinRand3}}$ .

If possible we simplify the arguments by linearly scaling the machine availabilities such that  $m = 1$  throughout this paper. Note that this does not change the competitive ratio  $\rho$ .

## 6 Competitive Analysis

We start with  $R = 3$ , the case we are most interested in.

**Lemma 2.**  $\rho_{\text{MinRand}} \geq 2/3$ .

*Proof.* The intuition of the proof is as follows: We define a non-decreasing function  $g$ , the argument of  $g$  is a non-negative real (a machine availability), and  $g$  returns a real. We define  $G := \sum_{a \in A} g(a)$ , where the set  $A$  is the availabilities of all machines.

In the first part of the proof we show that the **MinRand** algorithm freezes with  $G < N$ . In the second part of the proof we consider an optimal assignment of machines to files. If all files in the optimal assignment have availability strictly greater than  $3/2$ , we show that  $G \geq N$ . Since  $N \leq G < N$  is a contradiction, the minimum file of the optimal assignment has availability  $m^* \leq 3/2$ . With  $m = 1$  the proof will follow.

Here are the details: Let  $m = a_1 + a_2 + a_3$  be the file with minimum availability, with  $a_1 \geq a_2 \geq a_3$ . Of particular interest is  $a_3$ , the minimum-availability machine of minimum file  $m$ . Let  $g(a)$  be a function applied on the availability  $a$  of a machine, with

$$g(a) = \begin{cases} 1 & \text{if } a > 1 - a_3 \\ 1/2 & \text{if } 1/2 < a \leq 1 - a_3 \\ 1/4 & \text{if } a_3 < a \leq 1/2 \\ 0 & \text{if } a \leq a_3 \end{cases}$$

The function  $g(f)$  applied to file  $f$  is simply  $g(f) = g(b_1) + g(b_2) + g(b_3)$ , if file  $f = b_1 + b_2 + b_3$ .

Let  $f = b_1 + b_2 + b_3$  be another file with availability  $f > 1$ , and with  $b_1 \geq b_2 \geq b_3$ . We assume that there is no successful machine exchange between the files  $f$  and  $m$ . We denote the file  $f$  ( $m$ ) after an exchange with  $f'$  ( $m'$ ). Note that  $m' > 1$  and  $f' > 1$  would contradict the assumption that the **MinRand** algorithm froze, since  $f \geq m = 1$ .

We distinguish several cases.

Case 1: Let  $b_1 > 1 - a_3$ . If  $b_2 > a_3$  we can exchange the machines  $b_2$  and  $a_3$ , such that  $f' \geq b_1 + a_3 > (1 - a_3) + a_3 = 1$ , and  $m' = m + b_2 - a_3 > 1$ . Therefore  $b_2 \leq a_3$ . Thus  $g(f) = g(b_1) + g(b_2) + g(b_3) = 1 + 0 + 0 = 1$ .

In all other cases we therefore have  $b_1 \leq 1 - a_3$ .

Case 2: If  $b_3 \leq a_3$ , then  $g(f) \leq 1/2 + 1/2 + 0 = 1$ .

In all other cases we have  $b_3 > a_3$ .

Case 3: Let  $b_2 > 1/2$ . Since  $b_3 > a_3$  we can exchange the machines  $b_3$  and  $a_3$ , such that  $f' \geq b_1 + b_2 > 1$ , and  $m' = m + b_3 - a_3 > 1$ , which is a contradiction to the assumption that there was no successful exchange. Therefore we have  $b_2 \leq 1/2$ , and thus  $g(f) \leq 1/2 + 1/4 + 1/4 = 1$ .

So far we have shown that  $g(f) \leq 1$  for each file. A simple case study reveals that the minimum file  $m$  itself has  $g(m) \leq 3/4$ . Since each machine is part of exactly one file we have  $G = \sum_{f \in F} g(f)$ , where  $F$  is the set of all files. Since  $|F| = N$  we can conclude

$$G = \sum_{f \in F} g(f) \leq (N - 1) \cdot 1 + 3/4 < N.$$

In the second part of the proof we will show that if a file  $f$  has a sufficiently high availability, then  $g(f)$  will be at least 1. Specifially, we will show that for any file  $f = b_1 + b_2 + b_3$  (with  $b_1 \geq b_2 \geq b_3$ ) we have

$$f > 3/2 \Rightarrow g(f) \geq 1.$$

We distinguish two cases:

Case 1: If  $b_1 > 1 - a_3$  or  $b_2 > 1/2$  then  $g(f) \geq 1$ , because  $g(b_1) \geq 1$  or  $g(b_1) + g(b_2) \geq 1$ .

Case 2: We have  $b_1 \leq 1 - a_3$  and  $b_2 \leq 1/2$ . If  $b_1 \leq 1/2$ , then  $f = b_1 + b_2 + b_3 \leq 3/2$  (not satisfying the precondition that  $f > 3/2$ ). Thus  $1/2 < b_1 \leq 1 - a_3$  and  $b_2 \leq 1/2$ . We have  $3/2 < f = b_1 + b_2 + b_3 \leq (1 - a_3) + 1/2 + b_3 \Rightarrow b_3 > a_3$ . Then  $g(f) = 1/2 + 1/4 + 1/4 = 1$ .

In the second part of the proof we have shown that files  $f$  with availability  $f > 3/2$  necessarily have  $g(f) \geq 1$ .

The optimal algorithm assigns files to machines such that the file with minimum availability is  $m^*$ . Suppose, for the sake of contradiction, that an optimal algorithm manages to raise the availability of each file  $f$  such that  $f \geq m^* > 3/2$ . With the second part of the proof we know that in this case  $g(f) \geq 1$  for all  $N$  files. Since the function  $g$  of a file is defined as a sum of the function  $g$  of the machines of the file, we know that  $G \geq N$ . With the conclusion of the first part of the proof we get  $N \leq G < N$  which is a contradiction. Therefore  $m^* \leq 3/2$ , and  $\rho = m/m^* \geq 2/3$ .

**Lemma 3.**  $\rho_{\text{RandRand}3} \leq 2/3$ .

*Proof.* We give a constructive proof for a worst-case example with the three files  $m$ ,  $f_1$ , and  $f_2$ : The **RandRand** algorithm freezes with  $m = 1 + 0 + 0$  (the minimum-availability file),  $f_1 = 1 + 1 + 0$ , and  $f_2 = 1/2 + 1/2 + 1/2$ , that is, no machine exchange between any two files decreases the difference of the availabilities of the two files. We have nine machines with availabilities  $3 \times 1$ ,  $3 \times 1/2$ , and  $3 \times 0$ . An optimal algorithm generates three files  $1 + 1/2 + 0 = 3/2$ , thus  $m^* = 3/2$ . Therefore  $\rho_{\text{RandRand}3} = m/m^* \leq 2/3$ .

**Theorem 1.**  $\rho_{\text{MinRand}3} = \rho_{\text{RandRand}3} = 2/3$ .

*Proof.* The Theorem follows directly with the Lemmas [2](#), [3](#), and Lemma [1](#).

For replication factor 2 the **MinRand** algorithm is optimal:

**Theorem 2.**  $\rho_{\text{MinRand}2} = \rho_{\text{RandRand}2} = 1$ .

*Proof.* The proof is a “light” version of the proof of Lemma [2](#), and the details are omitted in this extended abstract. The function  $g$  is defined as

$$g(a) = \begin{cases} 1 & \text{if } a > 1 - a_2 \\ 1/2 & \text{if } a_2 < a \leq 1 - a_2 \\ 0 & \text{if } a \leq a_2 \end{cases}$$



The MinMax algorithm performs poorly in general, as the following example shows.

**Theorem 3.**  $\rho_{\text{MinMax}} = 0$ .

*Proof.* We give a constructive proof for a worst-case example with (at least) three files: Let  $m = 0 + 0 + 0 + \dots + 0$  be the file with minimum availability (note that  $m = 0$ ), and  $f = 3 + 0 + 0 + \dots + 0$  be the file with maximum availability, and all other files (at least one) have the machines  $1 + 1 + 0 + \dots + 0$ . The MinMax algorithm freezes since there is no exchange between the files  $m$  and  $f$ . For  $N \geq 3$  we have  $2(N - 2) + 1 \geq N$  machines with availability at least 1, and it is possible to reassign the machines to files such that each file has at least one machine with availability at least 1, that is  $m^* \geq 1$ . Thus  $\rho \leq 0/1 = 0$ .

We want to be confident that our algorithms do not fail with larger replication factors. In the following we give bounds on the performance for arbitrary  $R$ . All our bounds are tight for some  $R$ : As seen above, the MinMax algorithm is non-competitive for any  $R$ . The MinRand algorithm is worst when  $R \rightarrow \infty$ , and the RandRand algorithm is worst when  $R$  is 7.

**Lemma 4.** *Let  $m = a_1 + \dots + a_R$  be a file with availability  $m = 1$ , and  $a_1 \geq \dots \geq a_R \geq 0$ . Let  $f = b_1 + \dots + b_R > 1$  be another file, with  $1 \geq b_1 \geq \dots \geq b_R \geq 0$ . Let  $\bar{f}$  be  $f$ , but all the machines with availability less than  $a_R$  are replaced with machines with availability  $a_R$ , that is,  $\bar{f} = \max(b_1, a_R) + \max(b_2, a_R) + \dots + \max(b_R, a_R)$ . If  $\bar{f} > 2$  we can successfully exchange machines between  $m$  and  $f$ .*

*Proof.* Let  $l$  be the highest index such that  $b_l > a_R$ , that is, either  $b_{l+1} \leq a_R$  or  $l = R$ . First we exchange the machines  $b_l$  and  $a_R$ , that is  $m' = m + b_l - a_R > 1$  and  $f' = f + a_R - b_l$ . If  $f' > 1$  we are done, because we found a way to exchange machines and both availabilities are strictly greater than one. In the remainder of the proof we need to consider the case where  $f' \leq 1$  only.

As long as  $m' > 1$  and  $f' \leq 1$  we repeatedly exchange the machines  $a_{i+1}$  and  $b_{R-i}$ , for  $i = 0, 1, \dots$ . We denote  $m'$  ( $f'$ ) after the  $i$ th exchange with  $m^i$  ( $f^i$ ). More formally:

$$m^i = m' + \sum_{j=0}^i b_{R-j} - \sum_{j=0}^i a_{j+1} \text{ and } f^i = f' + \sum_{j=0}^i a_{j+1} - \sum_{j=0}^i b_{R-j}.$$

In the remainder of the proof we show that these repeated exchanges will eventually be successful, that is, there is a  $k$  (with  $k \leq R - l - 1$ ) such that after  $k$  exchanges we have  $m^k > m$  and  $f^k > m$ .

First, we show that the process of repeated exchanges *terminates*. In other words, there is a  $k \leq R - l - 1$  such that either  $m^k \leq 1$  or  $f^k > 1$ .

In particular, if  $i = R - l - 1$ , then

$$\begin{aligned}
 f^i &= f + a_R - b_l + \sum_{j=0}^i a_{i+1} - \sum_{j=0}^i b_{R-j} = \sum_{j=0}^{l-1} b_j + a_R + \sum_{j=0}^{R-l-1} a_{j+1} \\
 &\geq \sum_{j=0}^{l-1} \max(b_j, a_R) + a_R + \sum_{j=l+1}^R \max(a_R, b_j) \quad (\text{using } b_l > a_R \leq a_j) \\
 &= \bar{f} + a_R - b_l > 2 + a_R - b_l \geq 1. \quad (\text{using } b_l \leq 1)
 \end{aligned}$$

We have  $f^i > 1$ , and therefore the process of repeated exchanges terminates.

Since the process terminates after  $k$  exchanges, we have either  $m^k \leq 1$  or  $f^k > 1$ .

We distinguish three cases.

Case 1: Let  $m^k > 1$  and  $f^k > 1$ . We have found successful machine exchanges since  $m^k > m$  ( $m = 1$ ) and  $f^k > m$ . We are done.

Case 2: Let  $m^k \leq 1$  and  $f^k \leq 1$ . This contradicts the assumptions that  $m = 1$  and  $f > 1$  because  $2 \geq m' + f' = m + f > 2$ .

The only remaining case is the most difficult.

Case 3: Let  $m^k \leq 1$  and  $f^k > 1$ . Note that before the  $k$ th exchange it was decided to do another exchange, that is,  $m^{k-1} > 1$  and  $f^{k-1} \leq 1$ .

The precondition of this Lemma is  $\bar{f} = b_1 + \dots + b_{l-1} + b_l + (R-l)a_R > 2$ . For readability we split  $\bar{f}$  at “ $b_l$ ” into two parts:  $\bar{f} = \bar{f}_1 + \bar{f}_2$ .

Then

$$\begin{aligned}
 \bar{f}_1 &= b_1 + \dots + b_{l-1} = f - b_l - \sum_{i=l+1}^R b_i = f' - a_R - \sum_{i=l+1}^R b_i \\
 &= f' - a_R - \sum_{i=0}^{k-1} b_{R-i} - \sum_{i=l+1}^{R-k} b_i \leq f' - \sum_{i=0}^{k-1} b_{R-i} - a_R \\
 &= f^{k-1} - \sum_{i=0}^{k-1} a_{i+1} - a_R \leq 1 - \sum_{i=0}^{k-1} a_{i+1} - a_R
 \end{aligned}$$

Also we have

$$\begin{aligned}
 \bar{f}_2 &= b_l + (R-l)a_R \leq m + b_l - \sum_{i=1}^l a_i = m' + a_R - \sum_{i=1}^l a_i \\
 &= m^k + \sum_{i=0}^k a_{i+1} - \sum_{i=0}^k b_{R-i} + a_R - \sum_{i=1}^l a_i \leq 1 + a_R + \sum_{i=0}^k a_{i+1} - \sum_{i=1}^l a_i
 \end{aligned}$$

Together we get

$$\begin{aligned}
 \bar{f} &= \bar{f}_1 + \bar{f}_2 \leq 1 - \sum_{i=0}^{k-1} a_{i+1} - a_R + 1 + a_R + \sum_{i=0}^k a_{i+1} - \sum_{i=1}^l a_i \\
 &\leq 2 + a_k - a_1 \leq 2.
 \end{aligned}$$

This contradicts with the precondition  $\bar{f} > 2$ .

Only case 1 did not contradict with the assumptions and preconditions. The Lemma follows immediately.

**Lemma 5.**  $\rho_{\text{MinRand}} > 1/2$ .

*Proof.* Let  $m = a_1 + \dots + a_R = 1$  be a file with minimum availability, with  $a_1 \geq \dots \geq a_R \geq 0$ . Let  $f = b_1 + \dots + b_R > 1$  be another file, with  $b_1 \geq \dots \geq b_R \geq 0$ , and assume that there is no successful machine exchange.

We distinguish two types of files  $f$ :

Type A: Let  $b_1 > 1$ . If  $b_2 > a_R$  we can exchange the machines  $b_2$  and  $a_R$  such that  $m' = m + b_2 - a_R > 1$ ,  $f' \geq b_1 > 1$ . Therefore  $b_2 \leq a_R$ .

Type B:  $b_1 \leq 1$ .

Of the  $N$  files, one is the minimum file  $m$ ,  $x$  files are of type A,  $N - 1 - x$  files are of type B. We have exactly  $x$  machines with availability strictly greater than 1, that is, an optimal assignment will end up with at least  $N - x$  files that can only use machines with availability 1 or less. Since type A files have  $b_2 \leq a_R$  we can at most replace the machines of the type B files that have availability less than  $a_R$  with machines that have availability  $a_R$ . With Lemma 4 we know that such an improved file of type B can at most have availability 2. An optimal algorithm can redistribute the files such that the total sum of availabilities is at most  $G = 2(N - x - 1) + 1$ . An optimal redistribution to  $N - x$  files gives the new minimum file an availability of at most  $m^* \leq G/(N - x)$ , thus  $m^* < 2$ . Therefore  $\rho = m/m^* > 1/2$ .

**Theorem 4.**  $\lim_{R \rightarrow \infty} \rho_{\text{MinRand}_R} = 1/2$ .

*Proof.* For simplicity let  $R$  be a power of 2, that is  $R = 2^r$ . We construct the following files:

- The minimum file  $m = 1/R + \dots + 1/R = 1$ .
- $R/2$  files of type 0 with  $f = 2 + 1/R + 1/R + \dots + 1/R$ .
- For  $i = 1, \dots, r - 1$ :  $2^{r-i-1}$  files of type  $i$  with  $f = 2^i \times 2^{-i} + 0 + \dots + 0$

Note that there is no successful exchange between the minimum file  $m$  and any other file. We have used the following machines:

- $R/2$  machines with availability 2,
- For  $i = 1, \dots, r - 1$ :  $R/2$  machines with availability  $2^{-i}$ ,
- $R/2 \cdot (R - 1) + R$  machines with availability  $1/R$
- The rest of the machines have availability 0.

With the same machines we can build

- $R/2$  files of type A with  $f = 2 + \text{rest}$ , that is  $f \geq 2$ , and
- $R/2$  files of type B with  $f = 1/2 + 1/4 + 1/8 + \dots + 4/R + 2/R + 1/R + 1/R + \dots + 1/R$ , that is  $f = 1 + (R - r - 1)/R$ .

Since  $\lim_{R \rightarrow \infty} 1 + (R - r - 1)/R = 2$  we have  $m^* \rightarrow 2^-$ , and therefore  $\rho = m/m^* \rightarrow 1/2^+$ . With Lemma 5,  $\rho \rightarrow 1/2^+$  is tight and the Theorem follows.

**Lemma 6.**  $\rho_{\text{RandRand}} \geq 10/17$ .

*Proof.* We omit this tedious proof in the extended abstract. Apart from some complications it is similar to the proof of Lemma 2. The function  $g$  is defined as

$$g(a) = \begin{cases} 1 & \text{if } a > 1 \\ 1/2 & \text{if } 2/3 < a \leq 1 \\ a/(2-a) & \text{if } 1/2 < a \leq 2/3 \\ a/(1+a) & \text{if } a \leq 1/2 \end{cases}$$

**Theorem 5.**  $\rho_{\text{RandRand}} = 10/17$ , when we allow  $R$  to be a non-integer. If  $R$  must be an integer then  $\rho_{\text{RandRand}} \leq 3/5$ .

*Proof.* First we are going to make a relaxation to our normal model by assuming that  $R = 6 + \epsilon$ , where  $\epsilon > 0$ . (Remark: A non-integer  $R$  is an ethereal construct without physical realization; we need  $\epsilon$  to go towards 0 such that we can prove a tight bound.)

Let  $m = 1 + 0 + \dots$ . Additionally we have  $a$  files of type  $1 + 1 + 0 + \dots$ ,  $b$  files of type  $1/2 + 1/2 + 1/2 + 0 + \dots$ , and  $c$  files with  $6 + \epsilon$  machines with availability  $1/(5 + \epsilon)$ . Note that  $(6 + \epsilon)/(5 + \epsilon) = 1 + 1/(5 + \epsilon)$ . The RandRand algorithm does not find any successful exchange between any pair of files. If we do the accounting, we find  $2a + 1$  machines with availability 1,  $3b$  machines with availability  $1/2$ ,  $c(6 + \epsilon)$  machines with availability  $1/(5 + \epsilon)$ , and  $(5 + \epsilon) + a(4 + \epsilon) + b(3 + \epsilon)$  machines with availability 0.

An optimal algorithm assigns the machines so that it will get  $1 + a + b + c$  files of type  $f = 1 + 1/2 + 1/(5 + \epsilon) + 0 + \dots$  will have  $1 + a + b + c$  machines with availability 1,  $1/2$ , or  $1/(5 + \epsilon)$  each, and  $(1 + a + b + c)(3 + \epsilon)$  machines with availability 0. This is possible if and only if the following equation system is solvable:

$$\begin{aligned} 2a + 1 &= 1 + a + b + c \quad (\text{for availability } 1) \\ 3b &= 1 + a + b + c \quad (\text{for availability } 1/2) \\ c(6 + \epsilon) &= 1 + a + b + c \quad (\text{for availability } 1/(5 + \epsilon)) \\ (5 + \epsilon) + a(4 + \epsilon) + b(3 + \epsilon) &= (1 + a + b + c)(3 + \epsilon) \quad (\text{for availability } 0) \end{aligned}$$

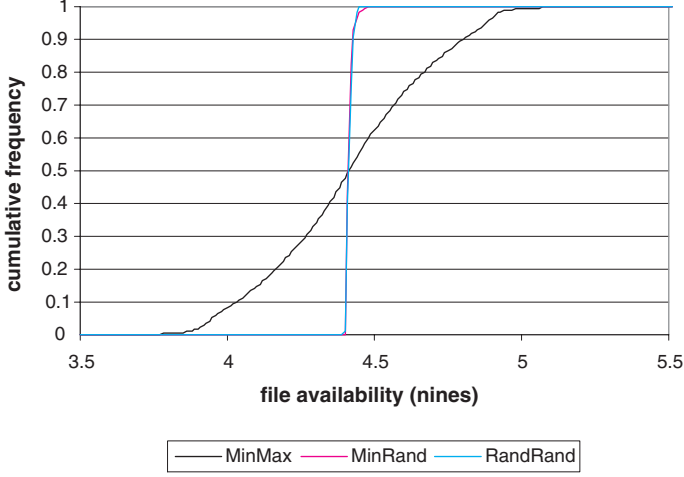
We can solve this equation system with  $a = 9/\epsilon + 1$ ,  $b = 6/\epsilon + 1$ ,  $c = 3/\epsilon$ . Thus we have a worst case example that is tight with Lemma 6.

$$m^* = f = \lim_{\epsilon \rightarrow 0^+} 1 + 1/2 + 1/(5 + \epsilon) = 17/10.$$

If  $R$  has to be integer we choose  $\epsilon = 1$ , and get  $m^* = 1 + 1/2 + 1/6 = 5/3$ .

## 7 Simulated Environment

We are given a set of  $M = 51,662$  machines with a measured distribution of availabilities in the range of 0.0 to 3.0 [9], calculated as the negative decimal logarithm of the machines' downtimes. (The common unit for availability is the “nine”; for example, a machine with a fractional downtime of 0.01 has  $-\log_{10} 0.01 = 2$  nines



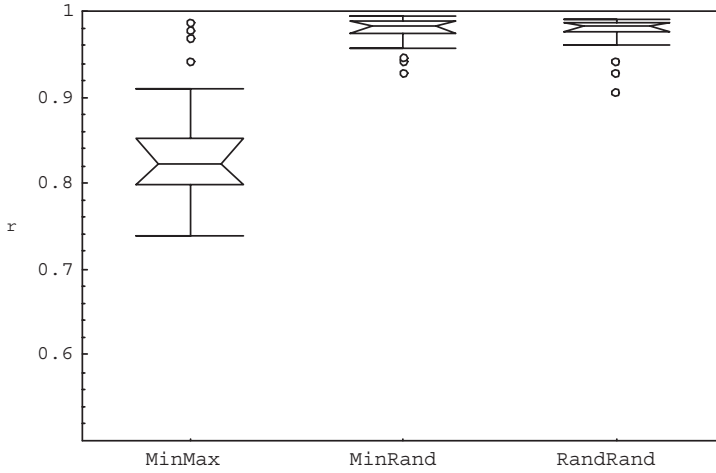
**Fig. 1.** File availability distributions

of availability, intuitively corresponding to its fraction uptime of  $1 - 0.01 = 0.99$ .) We are given a set of files whose sizes are governed by a binary lognormal distribution with  $m(2) = 12.2$  and  $s(2) = 3.43$  [7]. We simulate  $N = 2,583,100$  files, averaging 50 files per machine, which runs at the memory limit of the 512-MB computer we use for simulation. We maintain excess storage capacity in the system, without which it would not be possible to swap file replicas of different sizes. The mean value of this excess capacity is 10% of each machine’s storage space, and we limit file sizes to less than this mean value per machine. We fix the replication factor  $R = 3$  [9].

At each step, a pair of files is selected randomly (uniform distribution). To model the fact that in a distributed environment the selection of files for swapping is done without global knowledge, we set a selection range for minimum and maximum availability files 2%, to be consistent with a mean value of 50 files per machine. In other words, the “minimum-availability” file is drawn from the set of files with the lowest 2% of availabilities, and the “maximum-availability” file is drawn from the set of files with the highest 2% of availabilities, uniformly at random.

## 8 Simulation Results

We apply each of the algorithms to an initial random placement and run until the algorithm reaches a stable point. Fig. 1 shows file availability distributions for the three algorithms. The **MinMax** algorithm shows the widest variance, with an almost linear file availability distribution between 4 and 5 nines. The **RandRand**



**Fig. 2.** Minimum vs. Mean File Availability

algorithm yields a much tighter distribution, and the **MinRand** distribution is almost exactly the same as that for **RandRand**, except for the upper tail.

To study how the minimum file availability varies, we apply each of the algorithms to 100 randomly selected subsets of 100 machines from the measured set of 51,662 machines. Fig. 2 shows a box plot [13] of the ratio of the minimum file availability to the mean file availability for the three algorithms. The “waist” in each box indicates the median value, the “shoulders” indicate the upper quartile, and the “hips” indicate the lower quartile. The vertical line from the top of the box extends to a horizontal bar indicating the maximum data value less than the upper cutoff, which is the upper quartile plus  $3/2$  the height of the box. Similarly, the line from the bottom of the box extends to a bar indicating the minimum data value greater than the lower cutoff, which is the lower quartile minus  $3/2$  the height of the box. Data outside the cutoffs is represented as points.

The worst-case ratio that our simulation encountered for the **MinMax** algorithm is 0.74, which is poor but significantly better than the value of 0 which our competitive analysis showed is possible. The worst-case ratio found for **MinRand** is 0.93, and the worst-case ratio found for **RandRand** is 0.91. These values are both better than the theoretic competitive ratio of  $2/3$  for the two algorithms. Note that the simulation ratios use mean file availability as the denominator, rather than the optimum minimum file availability, since the latter is not easily computable. Therefore, the ratios may be artificially lowered by the possibly incorrect assumption that the mean file availability is an achievable value for the availability of the minimum file.

## 9 Related Work

Other than Farsite, serverless distributed file systems include xFS [2] and Frangipani [18], both of which provide high availability and reliability through distributed RAID semantics, rather than through replication. Archival InterMemory [5] and OceanStore [16] both use erasure codes and widespread data distribution to avoid data loss. The Eternity Service [1] uses full replication to prevent loss even under organized attack, but does not address automated placement of data replicas. A number of peer-to-peer file sharing applications have been released recently: Napster [17] and Gnutella [10] provide services for finding files, but they do not explicitly replicate files nor determine the locations where files will be stored. Freenet [6] performs file migration to generate or relocate replicas near their points of usage.

To the best of our knowledge this is the first study of the availability of replicated files. We know of negative results of hill-climbing algorithms in other areas, such as clustering [11].

There is a common denominator of our work and the research area of approximation algorithms, especially in the domain of online approximation algorithms [14,4] such as scheduling [12]. In online computing, an algorithm must decide how to act on incoming items without knowledge of the future. This is related to our work, in the sense that a distributed hill-climbing algorithm also makes decisions locally (without knowledge of the whole system), and where an adversary continuously changes the parameters of the system (e.g. the availabilities of the machines) in order to damage a good assignment of replicas to machines.

### Open Problems

There are a variety of questions we did not tackle in this paper. First, we focused on giving bounds for the efficacy of the three algorithms rather than efficiency. It is an interesting open problem how quickly the hill-climbing algorithms converge; both in a transient case (where we fix the availabilities of the machines and start with an arbitrary assignment of machines to files), and also in a steady-state case (where during an [infinite] execution of the algorithm an adversary with limited power can continuously change the availabilities of machines); see [8] for a simulation of the transient case and [9] for a simulation of the steady-state case.

It would also be interesting to drop some of the restrictions in this paper, in particular the simplification that each file has unit size.

Finally, it is an open problem whether there is another decentralized hill-climbing algorithm that has better efficacy and efficiency than the algorithms presented in this paper. For example, does it help if we considered exchanges between any group of three or four files? Or does it help to sometimes consider “downhill” exchanges too? In general, we would like to give lower bounds on the performance of *any* incremental and distributable algorithm. We feel that this area of research has a lot of challenging open problems, comparable with the depth and elegance of the area of online computation.

## References

1. Ross Anderson. The eternity service. *Proceedings of Pragocrypt*, 1996.
2. Thomas E. Anderson, Michael Dahlin, Jeanna M. Neefe, David A. Patterson, Drew S. Roselli, and Randolph Wang. Serverless network file systems. *ACM Transactions on Computer Systems*, 14(1):41–79, February 1996.
3. William J. Bolosky, John R. Douceur, David Ely, and Marvin Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop PCs. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computing Systems*, 2000.  
Also see <http://research.microsoft.com/sn/farsite/>.
4. Allan Borodin and Ran El-Yaniv. *Online Computation and Competitive Analysis*. Cambridge University Press, 1998.
5. Yuan Chen, Jan Edler, Andrew Goldberg, Allan Gottlieb, Sumeet Sobti, and Peter Yianilos. A prototype implementation of archival intermemory. In *Proceedings of the Fourth ACM International Conference on Digital Libraries*, 1999.
6. Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system, 2000.
7. John R. Douceur and William Bolosky. A large-scale study of file-system contents. In *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computing Systems*, pages 59–70, New York, May 1–4 1999.
8. John R. Douceur and Roger P. Wattenhofer. Large-scale simulation of replica placement algorithms for a serverless distributed file system. In *Proceedings of the 9th International Symposium on Modeling, Analysis and Simulation on Computer and Telecommunication Systems*, 2001.
9. John R. Douceur and Roger P. Wattenhofer. Optimizing file availability in a serverless distributed file system. In *Proceedings of the 20th Symposium on Reliable Distributed Systems*, 2001.
10. Gnutella. See <http://gnutelladev.wego.com>.
11. Nili Guttman-Beck and Refael Hassin. Approximation algorithms for min-sum p-clustering. In *Discrete Applied Mathematics*, vol. 89:1–3. Elsevier, 1998.
12. Leslie A. Hall. Approximation algorithms for scheduling. In Dorit S. Hochbaum, editor, *Approximation Algorithms for NP-Hard Problems*. PWS Publishing Company, 1995.
13. David M. Harrison. Mathematica experimental data analyst. *Wolfram Research, Champaign, IL*, 1996.
14. Sandy Irani and Anna R. Karlin. Online computation. In Dorit S. Hochbaum, editor, *Approximation Algorithms for NP-Hard Problems*. PWS Publishing Company, 1995.
15. Raj Jain. *The Art of Computer Systems Performance Analysis*. John Wiley & Sons, Inc., 1991.
16. John Kubiawicz, David Bindel, Patrick Eaton, Yan Chen, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Westley Weimer, Chris Wells, Hakim Weatherspoon, and Ben Zhao. OceanStore: An architecture for global-scale persistent storage. *ACM SIGPLAN Notices*, 35(11):190–201, November 2000.
17. Napster. See <http://www.napster.com>.
18. Chandramohan A. Thekkath, Timothy Mann, and Edward K. Lee. Frangipani: A scalable distributed file system. In *Proceedings of the 16th Symposium on Operating Systems Principles*, volume 31,5 of *Operating Systems Review*, pages 224–237, New York, October 5–8 1997.



# Optimal Unconditional Information Diffusion

Dahlia Malkhi, Elan Pavlov, and Yaron Sella

School of Computer Science and Engineering  
The Hebrew University of Jerusalem, Jerusalem 91904, Israel  
{dalia, elan, ysella}@cs.huji.ac.il

**Abstract.** We present an algorithm for propagating updates with information theoretic security that propagates an update in time logarithmic in the number of replicas and linear in the number of corrupt replicas. We prove a matching lower bound for this problem.

I cannot tell how the truth may be; I say the tale as 'twas said to me. – Sir Walter Scott

## 1 Introduction

In this paper, we consider the problem of secure information dissemination with information theoretic guarantees. The system we consider consists of a set of *replica* servers that store copies of some information, e.g., a file. A concern of deploying replication over large scale, highly decentralized networks is that some threshold of the replicas may become (undetectably) corrupt. Protection by means of cryptographic signatures on the data might be voided if the corruption is the action of an internal intruder, might be impossible if data is generated by low powered devices, e.g., replicated sensors, or might simply be too costly to employ. The challenge we tackle in this work is to spread *updates* to the stored information in this system efficiently and with unconditional security, while preventing corrupted information from contaminating good replicas. Our model is relevant for applications that employ a client-server paradigm with replication by the servers, for example distributed databases and quorum-systems.

More specifically, our problem setting is as follows. Our system consists of  $n$  replica servers, of which strictly less than a threshold  $b$  may be arbitrarily *corrupt*; the rest are *good* replicas. We require that each pair of good servers is connected by an authenticated, reliable, non-malleable communication channel. In order to be able to distinguish correct updates from corrupted (spurious) ones, we postulate that each update is initially input to an *initial set* of  $\alpha$  good replicas, where  $\alpha$  is at least  $b$ , the presumed threshold on the possible number of corrupt replicas. In a client-server paradigm, this means that the client's protocol for submitting an update to the servers addresses all the replicas in the initial set. The initial set is not known apriori, nor is it known to the replicas themselves at the outset of the protocol, or even during the protocol. Multiple updates are being continuously introduced to randomly designated initial sets,

and the diffusion of multiple updates actually occurs simultaneously. This is done by packing several updates in each message. Because we work with information theoretic security, the only criterion by which an update is accepted through diffusion by a good replica is when  $b$  different replicas independently vouch for its veracity. It should be stressed that we do not employ cryptographic primitives that are conditioned on any intractability assumptions, and hence, our model is the full Byzantine model without signatures.

The problem of secure information dissemination in a full Byzantine environment was initiated in [MMR99] and further explored in [MRRS01]. Because of the need to achieve information theoretic security, the only method to ascertain the veracity of updates is by replication. Consequently, those works operated with the following underlying principle: A replica is initially *active* for an update if it is input to it, and otherwise it is *passive*. Active replicas participate in a diffusion protocol to disseminate updates to passive replicas. A passive replica becomes active when it receives an update directly from  $b$  different sources, and consequently becomes *active* in its diffusion. For reasons that will become clear below, we call all algorithms taking this approach *conservative*. More formally:

**Definition 1.** A diffusion algorithm in which a good replica  $p$  sends an update  $u$  to another replica  $q$  only if  $p$  is sure of the update's veracity is called *conservative*.

In contrast, we call non-conservative algorithms *liberal*. Conservative algorithms are significantly limited in their performance. To illustrate this, we need to informally establish some terminology. First, for the purpose of analysis, we conceive of propagation protocols as progressing in synchronous *rounds*, though in practice, the rounds need not occur in synchrony. Further, for simplicity, we assume that in each round a good replica can send out at most one message (i.e., the *Fan-out*,  $F^{out}$ , is one); more detailed treatment can relate to  $F^{out}$  as an additional parameter. The two performance measures introduced in [MMR99] are as follows (precise definitions are given in the body of the paper):

- Let *Delay* denote the expected number of communication rounds from when an update is input to the system and until it reaches all the replicas;
- Let *Fan-in* ( $F^{in}$ ) denote the expected maximum number of messages received by any replica from good replicas in a round (intuitively, the  $F^{in}$  measures the “load” on replicas).

In [MMR99] a lower bound is shown on conservative algorithms of  $Delay * F^{in} = \Omega((nb/\alpha)^{1-\frac{1}{b}})$ . This linear lower bound is discouraging, especially compared with the cost of epidemic-style diffusion of updates in benign-failure environments<sup>1</sup>, which has  $Delay * F^{in} = O(\log n)$ . Such efficient diffusion would have been possible in a Byzantine setting if signatures were utilized to distinguish correct from spurious updates, but as already discussed, deploying digital signatures is ruled out in our setting. It appears that the advantages achieved by avoiding digital signatures come at a grave price.

<sup>1</sup> In epidemic-style diffusion we refer to a method whereby in each round, each active replica chooses a target replica independently at random and sends to it the update.

Fortunately, in this paper we propose an approach for diffusion in full Byzantine settings that is able to circumvent the predictions of [MMR99] using a fundamentally different approach. Our proposed liberal algorithm has  $\text{Delay} * F^{in} = O(b + \log n)$  and enjoys the same simplicity of epidemic-style propagation. The main price paid is in the size of messages used in the protocol. Although previous analyses ignored the size of messages, we note that our method requires additional communication space of  $n^{O(\log(b+\log n))}$  per message. In terms of delay, we prove our algorithm optimal by showing a general lower bound of  $\Omega(b^{\frac{n-\alpha}{n}} + \log \frac{n}{\alpha})$  on the delay for the problem model.

Our liberal approach works as follows. As before, a replica starts the protocol as *active* if it receives an update as input. Other replicas start as *passive*. Active replicas send copies of the update to other replicas at random. When a passive replica receives a copy of an update through another replica, it becomes *hesitant* for this update. A hesitant replica sends copies of the update, along with information about the paths it was received from, to randomly chosen replicas. Finally, when a replica receives copies of an update over  $b$  vertex-disjoint paths, it believes its veracity, and becomes active for it.

It should first be noted that this method does not allow corrupt updates to be accepted by good replicas. Intuitively, this is because when an update reaches a good replica, the last corrupt replica it passed through is correctly expressed in its path. Therefore, a spurious update cannot reach a good replica over  $b$  disjoint paths.

It is left to analyze the diffusion time and message complexity incurred by the propagation of these paths. Here, care should be taken. Since we show that a lower bound of  $\Omega(b^{\frac{n-\alpha}{n}} + \log \frac{n}{\alpha})$  holds on the delay, then if path-lengthening proceeds uncontrolled throughout the algorithm, then messages might carry up to  $O(b^b)$  paths. For a large  $b$ , this would be intolerable, and also too large to search for disjoint paths at the receivers. Another alternative that would be tempting is to try to describe the paths more concisely by simply describing the graph that they form, having at most  $O(nb)$  edges. Here, the problem is that corrupt replicas can in fact create spurious updates that appear to propagate along  $b$  vertex-disjoint paths in the graph, despite the fact that there were no such paths in the diffusion.

Our solution is to limit all paths to length  $\log \frac{n}{b}$ . That is, a replica that receives an update over a path of length  $\log \frac{n}{b}$  does not continue to further propagate this path. Nevertheless, we let the propagation process run for  $O(b + \log \frac{n}{b})$  rounds, during which paths shorter than  $\log \frac{n}{b}$  continue to lengthen. This process generates a dense collection of limited length paths. Intuitively, the diffusion process then evolves in two stages.

1. First, the diffusion of updates from the  $\alpha$  active starting points is carried as an independent epidemic-style process, so each one of the active replicas establishes a group of hesitant replicas to a vicinity of logarithmic diameter.
2. Each log-diameter vicinity of active replicas now directly targets (i.e., with paths of length 1) the remaining graph. With careful analyses it is shown that it takes additional  $O(b)$  rounds for each replica to be targeted directly

by some node from  $b$  out of the  $\alpha$  disjoint vicinities of active replicas, over  $b$  disjoint paths.

Throughout the protocol, each replica diffuses information about up to  $O((b + \log \frac{n}{b})^{\log \frac{n}{b}})$  different paths, which is the space overhead on the communication.

### 1.1 Related Work

Diffusion is a fundamental mechanism for driving replicated data to a consistent state in a highly decentralized system. Our work optimizes diffusion protocols in systems where arbitrary failures are a concern, and may form a basis of solutions for disseminating critical information in this setting.

The study of Byzantine diffusion was initiated in [MMR99]. That work established a lower bound for conservative algorithms, and presented a family of nearly optimal conservative protocols. Our work is similar to the approach taken in [MMR99] in its use of epidemic-style propagation, and consequently in its probabilistic guarantees. It also enjoys similar simplicity of deployment, especially in real-life systems where partially-overlapping universes of replicas exist for different data objects, and the propagation scheme needs to handle multiple updates to different objects simultaneously. The protocols of [MMR99] were further improved, and indeed, the lower bound of [MMR99] circumvented to some extent, in [MRRS01], but their general worst case remained the same.

The fundamental distinction between our work and the above works is in the liberal approach we take. With liberal approach, we are able to completely circumvent the lower bound of [MMR99], albeit at the cost of increased message size. An additional advantage of liberal methods is that in principle, they can provide update diffusion in any  $b$ -connected graph (though some topologies may increase the delay of diffusion), whereas the conservative approach might simply fail to diffuse updates if the network is not fully connected. The investigation of secure information diffusion in various network topologies is not pursued further in this paper however, and is a topic of our ongoing research. The main advantage of the conservative approach is that spurious updates generated by corrupt replicas cannot cause good replicas to send messages containing them; they may however inflict load on the good replicas in storage and in receiving and processing these updates. Hence, means for constraining the load induced by corrupt replicas must exist in both approaches.

While working on this paper, we learned that our liberal approach to secure information diffusion has been independently investigated by Minsky and Schneider [MS01]. Their diffusion algorithms use age to decide which updates to keep and which to discard, in contrast to our approach which discards based on the length of the path an update has traversed. Also, in the algorithms of [MS01], replicas pull updates, rather than push messages to other replicas, in order to limit the ability of corrupt hosts to inject bogus paths into the system. Simulation experiments are used in [MS01] to gain insight into the performance of those protocols; a closed-form analysis was sought by Minsky and Schneider but could not be obtained. Our work provides the foundations needed to analyze

liberal diffusion methods, provides general lower bounds, and proves optimality of the protocol we present.

Prior to the above works, previous work on update diffusion focused on systems that can suffer benign failures only. Notably, Demers et al. [DGH+87] performed a detailed study of epidemic algorithms for the benign setting, in which each update is initially known at a single replica and must be diffused to all replicas with minimal traffic overhead. One of the algorithms they studied, called *anti-entropy* and apparently initially proposed in [BLNS82], was adopted in Xerox’s Clearinghouse project (see [DGH+87]) and the Ensemble system [BHO+99]. Similar ideas also underly IP-Multicast [Dee89] and MUSE (for USENET News propagation) [LOM94]. The algorithms studied here for Byzantine environments behave fundamentally differently from any of the above settings where the system exhibits benign failures only.

Prior studies of update diffusion in distributed systems that can suffer Byzantine failures have focused on single-source broadcast protocols that provide reliable communication to replicas and replica agreement on the broadcast value (e.g., [LSP82, DS83, BT85, MR97]), sometimes with additional ordering guarantees on the delivery of updates from different sources (e.g., [Rei94, CASD95, MM95, KMM98, CL99]). The problem that we consider here is different from these works in the following ways. First, in these prior works, it is assumed that one replica begins with each update, and that this replica may be faulty—in which case the good replicas can agree on an arbitrary update. In contrast, in our scenario we assume that at least a threshold  $\alpha \geq b$  of good replicas begin with each update, and that only these updates (and no arbitrary ones) can be accepted by good replicas. Second, these prior works focus on reliability, i.e., *guaranteeing* that all good replicas (or all good replicas in some agreed-upon subset of replicas) receive the update. Our protocols diffuse each update to all good replicas only with some probability that is determined by the number of rounds for which the update is propagated before it is discarded. Our goal is to devise diffusion algorithms that are efficient in the number of rounds until the update is expected to be diffused globally and the load imposed on each replica as measured by the number of messages it receives in each round.

## 2 Preliminaries

Following the system model of [MMR99], our system consists of a universe  $S$  of  $n$  replicas to which updates are input. Strictly less than some known threshold  $b$  of the replicas could be *corrupt*; a corrupt replica can deviate from its specification arbitrarily (Byzantine failures). Replicas that always satisfy their specifications are *good*. We do not allow the use of digital signatures by replicas, and hence, our model is the full information-theoretic Byzantine model.

Replicas can communicate via a completely connected point-to-point network. Communication channels between good replicas are reliable and authenticated, in the sense that a good replica  $p_i$  receives a message on the communi-

cation channel from another good replica  $p_j$  if and only if  $p_j$  sent that message to  $p_i$ .

Our work is concerned with the diffusion of *updates* among the replicas. Each update  $u$  is introduced to an *initial set*  $I_u$  containing at least  $\alpha \geq b$  good replicas, and is then diffused to other replicas via message passing. Replicas in  $I_u$  are considered *active* for  $u$ . The goal of a diffusion algorithm is to make all good replicas *active* for  $u$ , where a replica  $p$  is active for  $u$  only if it can guarantee its veracity.

Our diffusion protocols proceed in synchronous rounds. For simplicity, we assume that each update arrives at each replica in  $I_u$  simultaneously, i.e., in the same round at each replica in  $I_u$ . This assumption is made purely for simplicity and does not impact on either the correctness or the speed of our protocol. In each round, each replica selects one other replica to which it sends information about updates as prescribed by the diffusion protocol. That is, the *Fan-out*,  $F^{out}$ , is assumed to be 1.<sup>2</sup> A replica receives and processes all the messages sent to it in a round before the next round starts.

We consider the following three measures of quality for diffusion protocols:

**Delay:** For each update, the delay is the worst-case expected number of rounds from the time the update is introduced to the system until all good replicas are active for update. Formally, let  $\eta_u$  be the round number in which update  $u$  is introduced to the system, and let  $\tau_p^u$  be the round in which a good replica  $p$  becomes active for update  $u$ . The delay is  $E[\max_{p,C} \{\tau_p^u\} - \eta_u]$ , where the expectation is over the random choices of the algorithm and the maximization is over good replicas  $p$ , all failure configurations  $C$  containing fewer than  $b$  failures, and all behaviors of those corrupt replicas. In particular,  $\max_{p,C} \{\tau_p^u\}$  is reached when the corrupt replicas send no updates, and our delay analysis applies to this case.

**Fan-in:** The fan-in measure, denoted by  $F^{in}$ , is the expected maximum number of messages that any good replica receives in a single round from good replicas under all possible failure scenarios. Formally, let  $\rho_p^i$  be the number of messages received in round  $i$  by replica  $p$  from good replicas. Then the fan-in in round  $i$  is  $E[\max_{p,C} \{\rho_p^i\}]$ , where the maximum is taken with respect to all good replicas  $p$  and all failure configurations  $C$  containing fewer than  $b$  failures. *Amortized fan-in* is the expected maximum number of messages received over multiple rounds, normalized by the number of rounds. Formally, a  $k$ -amortized fan-in starting at round  $l$  is  $E[\max_{p,C} \{\sum_{i=l}^{l+k} \rho_p^i / k\}]$ . We emphasize that fan-in and amortized fan-in are measures only for messages from good replicas.

**Communication complexity:** The maximum amount of information pertaining to a specific update, that was sent by a good replica in a single message. The maximum is taken on all the messages sent (in any round), and with respect to all good replicas and all failure configurations  $C$  containing fewer than  $b$  failures.

<sup>2</sup> We could expand the treatment here to relate to  $F^{out}$  as a parameter, but chose not to do so for simplicity.

Note that what interests us is the *expected* value of the measures. When we make statements of the type "within an expected  $f(r)$  rounds,  $P(r)$ " (for some predicate  $P$ , and function  $f$ ), we mean that if we define  $X$  as a random variable that measures the time until  $P(r)$  is true then  $E(X) = f(r)$ .

The following bound presents an inherent tradeoff between delay and fan-in for conservative diffusion methods (Definition [1](#)), when the initial set  $I_u$  is arbitrarily designated:

**Theorem 1** ([\[MMR99\]](#)). *Let there be a conservative diffusion algorithm. Denote by  $D$  the algorithm's delay, and by  $F^{in}$  its  $D$ -amortized fan-in. Then  $DF^{in} = \Omega(bn/\alpha)$ , for  $b \geq 2 \log n$ .*

One contribution of the present work is to show that the lower bound of Theorem [1](#) for conservative diffusion algorithms, does not hold once inactive replicas are allowed to participate in the diffusion.

### 3 Lower Bounds

In this section we present lower bounds which apply to any diffusion method in our setting. Our main theorem sets a lower bound on the delay. It states that the propagation time is related linearly to the number of corrupt replicas and logarithmically to the total number of replicas.

We start by showing the relation between the delay and the number of corrupt players.

**Lemma 1.** *Let there be any diffusion algorithm in our setting. Let  $D$  denote the algorithm's delay. Then  $D = \Omega(b \frac{n-\alpha}{n})$ .*

*Proof.* Since it is possible that there are  $b-1$  corrupt replicas, each good replica who did not receive the update initially as input must be targeted directly by at least  $b$  different other replicas, as otherwise corrupt replicas can cause it to accept an invalid update. Since only  $\alpha$  replicas receive the update initially, at least  $b(n-\alpha)$  direct messages must be sent. As  $F_{out} = 1$  and there are  $n$  replicas, at most  $n$  messages are sent in each round. Therefore it takes at least  $b \frac{n-\alpha}{n}$  rounds to have  $b(n-\alpha)$  direct messages sent.

We now show the relationship of the delay to the number of replicas.

**Lemma 2.** *Let there be any diffusion algorithm in our setting. Let  $D$  denote the algorithm's delay. Then  $D = \Omega(\log \frac{n}{\alpha})$ .*

*Proof.* Each replica has to receive a copy of the update. Since  $F_{out} = 1$ , the number of replicas who receive the update up to round  $t$  is at most twice the number of replicas who received the update up to round  $t-1$ . Therefore at the final round  $t_{end}$ , when all replicas received the update, we have that  $2^{t_{end}} \alpha = n$  or  $t_{end} = \log \frac{n}{\alpha}$ .

The following theorem immediately follows from the previous two lemmas:

**Theorem 2.** *Let there be any diffusion algorithm in our setting. Let  $D$  denote the algorithm's delay. Then  $D = \Omega(b^{\frac{n-\alpha}{n}} + \log \frac{n}{\alpha})$ .*

*Remark 1.* We will deal primarily in the case where  $\alpha \leq \frac{n}{2}$  as otherwise the diffusion problem is relatively simple. In particular, if  $\alpha > \frac{n}{2}$ , then we can use the algorithm of [MMR99] to yield delay of  $O(b)$ , which is optimal for  $F^{out} = 1$ . When  $\alpha \leq \frac{n}{2}$  our lower bound is equal to  $\Omega(b + \log \frac{n}{\alpha})$ , which is met by the propagation algorithm presented below.

*Remark 2.* We note that in order for an update to propagate successfully we must have that  $\alpha > b$ . From this, it immediately follows that  $b < \frac{n}{2}$ . However, below we shall have a tighter constraint on  $b$  that stems from our diffusion method. We note that throughout this paper no attempt is made to optimize constants.

## 4 The Propagation Algorithm

In this section we present an optimal propagation algorithm that matches the lower bound shown in section 3.

In our protocol, each replica can be in one of three states for a particular update: *passive*, *hesitant* or *active*. Each replica starts off either in the active state, if it receives the update initially as input, or (otherwise) in the passive state. In each round, the actions performed by a replica are determined by its state. The algorithm performed in a round concerning a particular update is as follows:

- 
- An active replica chooses a random replica and sends the update to it. (Compared with the actions of hesitant replicas below, the lack of any paths attached to the update conveys the replica's belief in the update's veracity.)
  - A passive or hesitant replica  $p$  that receives the update from  $q$ , with various (possibly empty) path descriptions attached, appends  $q$  to the end of each path and saves the paths. If  $p$  was passive, it becomes hesitant.
  - A hesitant replica chooses a random replica and sends to it all vertex-minimal paths of length  $< \log \frac{n}{b}$  over which the update was received.
  - A hesitant replica that has  $b$  vertex disjoint paths for the update becomes active.
- 

A couple of things are worth noting here. First, it should be clear that the algorithm above executes simultaneously for all concurrently propagating updates. Second, any particular update is propagated by replicas for a limited number of rounds. The purpose of the analysis in the rest of the paper is to determine the



number of rounds needed for the full propagation of an update. Finally, some optimizations are possible. For example, a hesitant replica  $p$  that has  $b$  vertex disjoint paths passing through a single vertex  $q$  (i.e., disjoint between  $q$  and  $p$ ) can unify the paths to be equivalent to a direct communication from the vertex  $q$ .

We now prove that our algorithm is correct.

**Lemma 3.** *If a good replica becomes active for an update then the update was initially input to a good replica.*

*Proof.* There are two possible ways in which a good replica can become active for an update. The first possibility is when the replica receives the update initially as input. In this case the claim certainly holds.

The second possibility is when the replica receives the update over  $b$  vertex disjoint paths. We say that a corrupt replica *controls* a path if it is the last corrupt replica in the path. Note that for any invalid update which was generated by corrupt replica(s), there is exactly one corrupt replica controlling any path (since by definition the update was created by the corrupt replicas). Since good replicas follow the protocol and do not change the path(s) they received, the corrupt controlling replica will not be removed from any path by any subsequent good replica receiving the update. As there are less than  $b$  corrupt replicas and the paths are vertex disjoint there are less than  $b$  such paths. As a good replica becomes active for an update when it receives the update over  $b$  disjoint paths, at least one of the paths has only good replicas in it. Therefore the update was input to a good replica.

The rest of this paper will prove the converse direction. If an update was initially input to  $\alpha \geq b$  good replicas then within a relatively small number of rounds, all good replicas will receive the update with high probability.

## 5 Performance Analysis

In this section, we proceed to analyze the performance of our algorithm. Our treatment is based on a communication graph that gradually evolves in the execution. We introduce some notation to be used in the analysis below. At every round  $r$ , the communication graph  $G_r = (V, E_r)$  is defined on (good) vertices  $V$  such that there is a (directed) edge between two vertices if one sent any message to the other during round  $r$ . We denote by  $N_G(I)$  the neighborhood of  $I$  (singleton or set) in  $G$ . We denote by  $\|p, q\|_G$  the shortest distance between  $p$  and  $q$  in  $G$ . In the analysis below, we use vertices and replicas interchangeably.

Our proof will make use of *gossip-circles* that gradually evolve around active replicas. Intuitively, the gossip-circle  $C(p, d, r)$  of a good active replica is the set of good replica that heard the update from  $p$  over good paths (comprising good replicas) of length up to  $d$  in  $r$  rounds. Formally:

**Definition 2.** Let  $p$  be some good replica which is active for the update  $u$ . Let  $\{G_j = (V, E_j)\}_{j=1..r}$  be the set of communication graphs of  $r$  rounds of the execution of vertices in  $V$ . Recall that  $N_G(I)$  denotes the set of all neighbors in a graph  $G$  of nodes in  $I$ . We then define gossip circles of  $p$  in  $r$  rounds inductively as follows:

$$C_V(p, 0, r) = \{p\}$$

$$\forall 1 \leq d \leq r :$$

$$C_V(p, d, r) = C_V(p, d-1, r) \cup$$

$$\{q \in N_{G_d}(C_V(p, d-1, r)) : \|p, q\|_{C_V(p, d-1, r)} \leq \min\{d-1, \log \frac{n}{b} - 1\}\}$$

When  $V$  is the set of good replicas, we omit it for simplicity. Note that the gossip circle  $C(p, d, r)$  is constrained by definition to have radius  $\leq \min\{d, \log \frac{n}{b}\}$ .

The idea behind our analysis is that any  $b$  initial active good replicas spread paths that cover *disjoint* low-diameter gossip-circles of size  $\frac{n}{4b}$ . Hence, it is sufficient for any replica to be directly targeted by some replica from each one of these sets in order to have  $b$  vertex-disjoint paths from initial replicas.

We first show a lemma about the spreading of epidemic style propagation with bounded path length. Without bounding paths, the analysis reduces to epidemic-style propagation for benign environment, as shown in [DGH+87].

**Lemma 4.** Let  $p \in I_u$  be a good replica, and let  $d \leq \log \frac{n}{b}$ . Assume there are no corrupt replicas. Then within an expected  $r > d$  rounds,  $|C(p, d, r)| \geq \min\{(\frac{3}{2})^d + (r-d)(\frac{3}{2})^{d-4}, \frac{n}{2}\}$ .

*Proof.* The proof looks at an execution of  $r$  rounds of propagation in two parts. The first part consists of  $d$  rounds. In this part, the set of replicas that received a copy of  $u$  (equivalently, received a copy of  $u$  over paths of length  $\leq d$ ), grows exponentially. That is, in  $d$  rounds, the update propagates to  $(\frac{3}{2})^d$  replicas. The second part consists of the remaining  $r-d$  rounds. This part makes use of the fact that at the end of the first part, an expected  $(\frac{3}{2})^{d-4}$  replicas receive a copy of  $u$  over paths of length  $< d$ . Hence, in the second part, a total of  $(r-d) \times (\frac{3}{2})^{d-4}$  replicas receive  $u$ .

Formally, let  $m_j$  denote the number of replicas that received  $u$  from  $p$  over paths of length  $\leq d$  by round  $j$ , i.e.,  $m_j = |C(p, d, j)|$ .

Let  $j \leq d$ . So long as the number of replicas reached by paths of length  $\leq d$  does not already exceed  $\frac{n}{2}$ , then in round  $j+1$  each replica in  $C(p, d, j)$  targets a new replica with probability  $\geq \frac{1}{2}$ . Therefore, the expected number of messages sent until  $\frac{m_j}{2}$  new replicas are targeted is at most  $m_j$ . Furthermore, since at least  $m_j$  messages are sent in round  $j$ , this occurs within an expected one round. We therefore have that the expected time until  $(\frac{3}{2})^d$  replicas receive  $u$  over paths of length  $\leq d$  is at most  $d$ .

From round  $d+1$  on, we note that at least half of  $m_d$  received  $u$  over paths of length strictly less than  $d$ . Therefore, in each round  $j > d$ , there are at least  $\frac{1}{2} \times (\frac{3}{2})^d$  replicas forwarding  $u$  over paths of length  $< d$ . So long as  $m_j \leq \frac{n}{2}$ , then in round  $j$  each of these replicas targets a new replica with probability  $\geq \frac{1}{2}$ . Therefore, the expected number of messages sent until  $(\frac{3}{2})^{d-4} < \frac{1}{2} \times \frac{1}{2} \times (\frac{3}{2})^d$

new replicas are targeted is at most  $\frac{1}{2} \times (\frac{3}{2})^d$ , which occurs in an expected one round.

Putting the above together, we have that within an expected  $r$  rounds,  $(\frac{3}{2})^d + (r - d) \times (\frac{3}{2})^{d-4}$  replicas are in  $C(p, d, r)$ .

Since the choice of communication edges in the communication graph is made at random, we get as an immediate corollary:

**Corollary 1.** *Let  $V' \subseteq V$  be a set of vertices, containing all corrupt ones, chosen independently from the choices of the algorithm, such that  $|V'| \leq \frac{n}{3}$ . Let  $p \in I_u$  be a good replica, and let  $d \leq \log \frac{n}{b}$ . Then within an expected  $3r > d$  rounds,  $|C_{(V \setminus V')}(p, d, 3r)| \geq \min\{(\frac{3}{2})^d + (r - d)(\frac{3}{2})^{d-4}, \frac{n}{2}\}$ .*

We now use corollary [1](#) to build  $b$  disjoint gossip circles of initial replicas, and wish to proceed with the analysis of the number of rounds it takes for replicas to be targeted by these disjoint sets. As edges in the communication graph are built at random, a tempting approach would be to treat this as a simple coupon collector problem on the  $b$  gossip-circles where each replica wishes to “collect a member” of each of these sets by being targeted with an edge from it. With this simplistic analysis, it would take each replica  $O(b \log b)$  rounds to collect all the coupons, and an additional logarithmic factor in  $n$  for all replicas to complete. The resulting analysis would provide an upper bound of  $O(b(\log b)(\log n))$  on the delay. Although this is sufficient for small  $b$ , for large  $b$  we wish to further tighten the analysis on the number of rounds needed for diffusion.

The approach we take is to gradually adapt the size of the disjoint gossip-circles as the process evolves, and to show that the expected amount of time until all sets are connected to a replica remains constant. More precisely, we show that in an expected  $O(b)$  rounds, a replica has edges to half of  $b$  gossip-circles of size  $\frac{n}{4b}$ . We then look at the communication graph with all of the vertices in the paths of the previous step(s) removed. We show that in time  $O(b/2)$ , a replica has edges to gossip-circles of size  $\frac{2n}{4b}$  of half of the  $\frac{b}{2}$  remaining initial replicas. And so on. In general, we have an inductive analysis for  $k = 0.. \log b$ . For each  $k$ , we denote  $b_k = \frac{b}{2^k}$ . For step  $k$  of the analysis, we show that in time  $O(b_k)$ , a replica has disjoint paths of length  $\leq \log \frac{n}{b \times b_k}$  to  $\frac{b_k}{2}$  of the initial replicas. Hence, in total time  $O(b)$ , a replica connects to  $b$  initial replicas over disjoint paths, all of length  $\leq \log \frac{n}{b}$  (and hence, not exceeding the algorithm’s path limit).

Our use of Corollary [1](#) is as follows. Let  $b_k = \frac{b}{2^k}$ , and let  $V'$  denote a set of vertices we wish to exclude from the graph, where  $|V'| \leq \frac{n}{3}$ . Then we have that within an expected  $3r = 3(b + 2 \log \frac{n}{b \times b_k})$  rounds, each initial good replica has a gossip circle of diameter  $d = \max\{1, 2 \log \frac{n}{b \times b_k}\}$  whose size is at least  $(b + d - d)(\frac{3}{2})^{(d-4)} \geq \frac{n}{4b_k}$ .

We now use this fact to designate disjoint low-diameter gossip circles around  $b$  good replicas in  $I_u$ .

**Lemma 5.** *Let  $I \subseteq I_u$  be a subset of initial good replicas of size  $b_k$ . Let  $W'$  be a subset of replicas with  $|W'| \leq \frac{n}{12}$ . Denote by  $d = \max\{1, 2 \log \frac{n}{b \times b_k}\}$ . Then within an expected  $3r = 3(b + d)$  rounds there exist disjoint subsets  $\{C_i\}_{i \in I}$  containing no vertices of  $W'$ , such that each  $C_i \subseteq C_{(V \setminus W')}(i, d, 3r)$ , and such that each  $|C_i| = \frac{n}{4b_k}$ .*

*Proof.* The proof builds these sets for  $I$  inductively. Suppose that  $C_1, \dots, C_{i-1}$ , for  $0 < i \leq b_k$ , have been designated already, such that for all  $1 \leq j \leq i-1$ , we have that  $C_j \subseteq C(j, d, 3r)$  and  $|C_j| = \frac{n}{4b_k}$ . Denote by  $C' = \bigcup_{j=1..i-1} C_j$ . Then the total number of vertices in  $V' = C' \cup W'$  is at most  $\frac{n}{12} + (i-1) \frac{n}{4b_k} \leq \frac{n}{12} + b_k \frac{n}{4b_k} \leq \frac{n}{3}$ . From Corollary [1](#), we get that within an expected  $3r$  rounds, and without using any vertex in  $V'$ , the gossip circle  $C_{V \setminus V'}(i, d, 3r)$  contains at least  $\left(b + 2 \log \frac{n}{b \times b_k} - 2 \log \frac{n}{b \times b_k}\right) \left(\frac{3}{2}\right)^{\left(2 \log \frac{n}{b \times b_k} - 4\right)} \geq \frac{n}{4b_k}$ . Hence, we set  $C_i$  to be a subset of  $C(i, d, 3r)$  of size  $\frac{n}{4b_k}$  and the lemma follows.

We now analyze the delay until a vertex has direct edges to these  $b_k$  disjoint sets.

**Lemma 6.** *Let  $v \in V$  be a good replica. Let  $b_k = \frac{b}{2^k}$  as before and let  $\{C_i\}_{i=1..b_k}$  be disjoint sets, each of size  $\frac{n}{4b_k}$  and diameter  $2 \log \frac{n}{b \times b_k}$  (as determined by Lemma [5](#)). Then within an expected  $4b_k$  rounds there are edges from  $\frac{b_k}{2}$  of the sets to  $v$ .*

*Proof.* The proof is simply a coupon collector analysis of collecting  $\frac{b_k}{2}$  out of  $b_k$  coupons, where in epoch  $i$ , for  $1 \leq i \leq \frac{b_k}{2}$ , the probability of collecting the  $i$ 'th new coupon in a round is precisely the probability of  $v$  being targeted by a new set, i.e.,  $\frac{(b_k - i) \frac{n}{4b_k}}{n}$ . The expected number of rounds until completion is therefore  $\sum_{i=1..(b_k/2)} \frac{4b_k}{b_k - i} \leq 4b_k$ .

We are now ready to put these facts together to analyze the delay that a single vertex incurs for having disjoint paths to  $b$  initial replicas.

**Lemma 7.** *Let  $v \in V$  be a good replica. Suppose that  $b < \frac{n}{60}$ . Then within an expected  $5(b + \log \frac{n}{b})$  rounds there are  $b$  vertex disjoint paths of length  $\leq \log \frac{n}{b}$  from  $I_u$  to  $v$ .*

*Proof.* We prove by induction on  $b_k = \frac{b}{2^k}$ , for  $k = 0..(\log b - 1)$ . To begin the induction, we set  $b_0 = b$ . By Corollary [1](#) within an expected  $b + 2 \log \frac{n}{b \times b_0}$  stages, there are  $b_0 = b$  disjoint sets (of radius  $2 \log \frac{n}{b \times b_0}$ ) whose size is  $\frac{n}{4b_0}$ . By Lemma [6](#), within  $4b_0$  rounds,  $v$  has direct edges to  $\frac{b_0}{2}$  of these sets. Hence, it has disjoint paths of length  $\leq 2 \log \frac{n}{b \times b_0} + 1$  to  $\frac{b_0}{2}$  initial replicas. These paths comprise at most  $\frac{b_0}{2} (2 \log \frac{n}{b \times b_0} + 1)$  good vertices.

For step  $0 \leq k < (\log b)$  of the analysis, we set  $b_k = \frac{b}{2^k}$ . The set of vertices used in paths so far, together with all the corrupt vertices, total less than

$$b + \sum_{k' < k} \frac{b_{k'}}{2} \left( 2 \log \frac{n}{b \times b_{k'}} + 1 \right) \leq b + \sum_{k' < k} \frac{b}{2^{k'}} \left( \log \frac{2^{k'} n}{b^2} + 1 \right) \leq b + 2b(1 + \log \frac{n}{b^2}).$$

By our assumption that  $b < \frac{n}{60}$ , we get that the total number of vertices used until step  $k$  is less than  $\frac{n}{12}$ . Hence, in each step  $0 \leq k < \log b$ , we apply Corollary 1 to form  $b_k$  disjoint sets (of radius  $2 \log \frac{n}{b \times b_k}$ ) whose size is  $\frac{n}{4b_k}$  each. By Lemma 6, half of these sets have direct edges to  $v$  within an expected  $4b_k$  rounds.

In total, we showed that in expected  $\max_{0 \leq k < \log b} \{4b_k + b + 2 \log \frac{n}{b \times b_k}\}$  rounds,  $v$  has disjoint paths (of length at most  $\log \frac{n}{b}$ ) to  $b$  initial replicas.

We now wish to bound the time when **all** of the nodes have  $b$  vertex disjoint paths to  $I_u$ . A tempting approach would be to use a Chernoff bound, but the analysis would then require an additional logarithmic factor in  $n$ . This factor can be avoided by utilizing the fact that after a  $O(\log n + b)$  rounds there exist a fraction of the replicas who are active for the update. Finally, propagation from a linear set is easily done.

**Lemma 8.** *Let  $c > 1$  be a constant. The expected time until  $(n - b)(1 - \frac{1}{c})$  replicas become active is  $O(b + \log n)$ .*

*Proof.* By Lemma 7, the expected time for a replica to become active is  $5(b + \log \frac{n}{b})$ . Hence, the probability that a replica becomes active in  $c \times 5(b + \log \frac{n}{b})$  rounds or more is less than  $\frac{1}{c}$ . Hence, within an expected  $c \times 5(b + \log \frac{n}{b})$  rounds the number of active replicas is at least  $(n - b)(1 - \frac{1}{c})$ .

We now choose a particular value for  $c$  in the previous lemma. We note that we choose an arbitrary value without attempting to minimize the constants.

For  $c = 2$ , within an expected  $10(b + \log \frac{n}{b})$  rounds there are  $\frac{1}{2}(n - b)$  replicas who are active for the update. By reusing the supposition  $b < \frac{n}{60}$  from Lemma 7, we get that  $\frac{1}{2}(n - b) > \frac{1}{2}(n - \frac{n}{60}) > \frac{2}{5}n$ . This means that there are at least  $\frac{2}{5}n$  good replicas who are active for the update.

**Lemma 9.** *If at least  $\frac{2}{5}n$  good replicas are active for the update then within an expected  $O(b + \log n)$  rounds all of the replicas become active for the update.*

*Proof.* Fix any replica and let  $Y_i$  be the number of updates from active replicas that the replica receives in round  $i$ . Let  $Y$  be the number of updates that the replica receives in  $r$  rounds, i.e.,  $Y = \sum_{i=1}^r Y_i$ . By the linearity of expectation,  $E(Y) = \sum_{i=1}^r E(Y_i) \geq \frac{2}{5}r$ . Using a Chernoff bound we have that  $Pr[Y \leq \frac{r}{10}] \leq e^{-\frac{r}{48}}$ . Therefore if  $r = 48 \log n + 2b$  we have that  $Pr[Y \leq \frac{r}{10}] \leq \frac{1}{n^2}$ .

**Theorem 3.** *The algorithm terminates in an expected  $O(\log n + b)$  rounds.*

*Proof.* By corollary 7 and lemma 8 it follows that within  $O(\log n + b)$  rounds  $0.8$  of the replicas become active. From Lemma 9 within an additional  $O(\log n + b)$  rounds all of the replicas become active.

Therefore, our delay matches the lower bound of theorem 2.

We conclude the analysis with a log amortized  $F^{in}$  analysis and a communication complexity bound. The  $\log n$  amortized  $F^{in}$  of our algorithm as shown in [MMR99] is 1.

In order to finish the analysis the communication complexity (which also bounds the required storage size) must be addressed. Each vertex  $v \in V$  receives at most  $O(b + \log \frac{n}{b})$  sets of paths. Paths are of length at most  $\log \frac{n}{b}$ . Therefore, the communication overhead per message can be bounded by  $O(b + \log \frac{n}{b})^{\log \frac{n}{b}} = (\frac{n}{b})^{O(\log(b + \log n))}$ .

This communication complexity can be enforced by good replicas even in the presence of faulty replicas. A good replica can simply verify that (a) the length of all paths in any incoming message does not exceed  $\log \frac{n}{b}$ , and that (b) the out-degree of any vertex does not exceed  $O(b + \log \frac{n}{b})$ . Any violation of (a) or (b) indicates that the message was sent by a faulty replica, and can be safely discarded.

## 6 Conclusions and Future Work

This paper presented a round-efficient algorithm for disseminating updates in a Byzantine environment. The protocol presented propagates updates within an expected  $O(b + \lg n)$  rounds, which is shown to be optimal. Compared with previous methods, the efficiency here was gained at the cost of an increase in the size of messages sent in the protocol. Our main direction for future work is to reduce the communication complexity, which was cursorily addressed in the present work.

## References

- [BHO+99] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budio and Y. Minsky. Bimodal multicast. *ACM Transactions on Computer Systems* 17(2):41–88, 1999.
- [BLNS82] A. D. Birrell, R. Levin, R. M. Needham, and M. D. Schroeder. Grapevine, An exercise in distributed computing. *Communications of the ACM* 25(4):260–274, 1982.
- [BT85] G. Bracha and S. Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM* 32(4):824–840, October 1985.
- [CL99] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation*, 1999.
- [CASD95] F. Cristian, H. Aghili, R. Strong, and D. Dolev. Atomic broadcast: From simple message diffusion to Byzantine agreement. *Information and Computation* 18(1), pages 158–179, 1995.
- [Dee89] S. E. Deering. Host extensions for IP multicasting. SRI Network Information Center, RFC 1112, August 1989.
- [DGH+87] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic algorithms for replicated database maintenance. In *Proceedings of the 6th ACM Symposium on Principles of Distributed Computing*, pages 1–12, 1987.
- [DS83] D. Dolev and R. Strong. Authenticated algorithms for Byzantine agreement. *SIAM Journal of Computing* 12(4):656–666, 1983.

- [KMM98] K. P. Kihlstrom, L. E. Moser and P. M. Melliar-Smith. The SecureRing protocols for securing group communication. In *Proceedings of the 31st IEEE Annual Hawaii International Conference on System Sciences*, vol. 3, pages 317–326, January 1998.
- [LOM94] K. Lidl, J. Osborne and J. Malcome. Drinking from the firehose: Multicast USENET news. In *Proceedings of the Usenix Winter Conference*, pages 33–45, January 1994.
- [LSP82] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems* 4(3):382–401, July 1982.
- [MM95] L. E. Moser and P. M. Melliar-Smith. Total ordering algorithms for asynchronous Byzantine systems. In *Proceedings of the 9th International Workshop on Distributed Algorithms*, Springer-Verlag, September 1995.
- [MMR99] D. Malkhi, Y. Mansour, and M. K. Reiter. On diffusing updates in a Byzantine environment. In *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems*, pages 134–143, October 1999.
- [MR97] D. Malkhi and M. Reiter. A high-throughput secure reliable multicast protocol. *Journal of Computer Security* 5:113–127, 1997.
- [MRRS01] D. Malkhi, M. Reiter, O. Rodeh and Y. Sella. Efficient update diffusion in Byzantine environments. To appear in *Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems*, 2001.
- [MS01] Y. Minsky and F. B. Schneider. Tolerating malicious gossip. Private communication.
- [Rei94] M. K. Reiter. Secure agreement protocols: Reliable and atomic group multicast in Rampart. In *Proceedings of the 2nd ACM Conference on Computer and Communications Security*, pages 68–80, November 1994.

# Computation Slicing: Techniques and Theory

Neeraj Mittal<sup>1</sup> and Vijay K. Garg<sup>2\*</sup>

<sup>1</sup> Department of Computer Sciences

The University of Texas at Austin, Austin, TX 78712, USA

neerajm@cs.utexas.edu <http://www.cs.utexas.edu/users/neerajm>

<sup>2</sup> Department of Electrical and Computer Engineering

The University of Texas at Austin, Austin, TX 78712, USA

garg@ece.utexas.edu <http://www.ece.utexas.edu/~garg>

**Abstract.** We generalize the notion of slice introduced in our earlier paper [6]. A slice of a distributed computation with respect to a global predicate is the smallest computation that contains all consistent cuts of the original computation that satisfy the predicate. We prove that slice exists for all global predicates. We also establish that it is, in general, NP-complete to compute the slice. An optimal algorithm to compute slices for special cases of predicates is provided. Further, we present an efficient algorithm to graft two slices, that is, given two slices, either compute the smallest slice that contains all consistent cuts that are common to both slices or compute the smallest slice that contains all consistent cuts that belong to at least one of the slices. We give application of slicing in general and grafting in particular to global property evaluation of distributed programs. Finally, we show that the results pertaining to consistent global checkpoints [14,18] can be derived as special cases of computation slicing.

## 1 Introduction

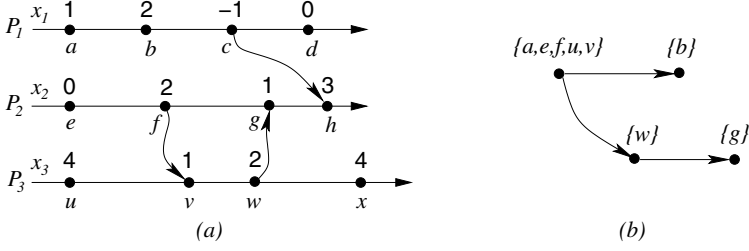
Writing distributed programs is an error prone activity; it is hard to reason about them because they suffer from the combinatorial explosion problem. Testing and debugging, and software fault-tolerance is an important way to ensure the reliability of distributed systems. Thus it becomes necessary to develop techniques that facilitate the analysis of distributed computations. Various abstractions such as predicate detection (e.g., [13,7]) and predicate control [16,17,11] have been defined to carry out such analysis.

In our earlier paper [6], we propose another abstraction, called *computation slice*, which was defined as: a slice of a distributed computation with respect to a global predicate is another computation that contains *those and only those* consistent cuts (or snapshots) of the original computation that satisfy the predicate. In [6], we also introduce a class of global predicates called *regular predicates*: a global predicate is regular iff whenever two consistent cuts satisfy the predicate

---

\* supported in part by the NSF Grants ECS-9907213, CCR-9988225, Texas Education Board Grant ARP-320, an Engineering Foundation Fellowship, and an IBM grant.





**Fig. 1.** (a) A computation and (b) its slice with respect to  $(x_1 \geq 1) \wedge (x_3 \leq 3)$ .

then the cuts given by their set intersection and set union also satisfy the predicate. We show that slice exists only for regular predicates and present an efficient algorithm to compute the slice. The class of regular predicates is closed under conjunction.

A limitation of the definition of slice in [6] is that slice exists only for a specific class of predicates. This prompted us to weaken the definition of slice to the *smallest* computation that contains all consistent cuts of the original computation that satisfy the predicate. In this paper, we show that slice exists for all global predicates.

The notion of computation slice is analogous to the concept of program slice [19]. Given a program and a set of variables, a program slice consists of all statements in the program that may affect the value of the variables in the set at some given point. A slice could be static [19] or dynamic (for a specific program input) [9]. The notion of a slice has been also extended to distributed programs [8]. Program slicing has been shown to be useful in program debugging, testing, program understanding and software maintenance [9, 19]. A slice can significantly narrow the size of the program to be analyzed, thereby making the understanding of the program behaviour easier. We expect to reap the same benefit from a computation slice.

Computation slicing is also useful for reducing search space for NP-complete problems such as predicate detection [3, 7, 15, 13]. Given a distributed computation and a global predicate, predicate detection requires finding a consistent cut of the computation, if it exists, that satisfies the predicate. It is a fundamental problem in distributed system and arises in contexts such as software fault tolerance, and testing and debugging.

As an illustration, suppose we want to detect the predicate  $(x_1 * x_2 + x_3 < 5) \wedge (x_1 \geq 1) \wedge (x_3 \leq 3)$  in the computation shown in Fig. 1(a). The computation consists of three processes  $P_1$ ,  $P_2$  and  $P_3$  hosting integer variables  $x_1$ ,  $x_2$  and  $x_3$ , respectively. The events are represented by solid circles. Each event is labeled with the value of the respective variable immediately after the event is executed. For example, the value of variable  $x_1$  immediately after executing the event  $c$  is  $-1$ . The first event on each process initializes the state of the process and every consistent cut contains these initial events. Without computation slicing,

we are forced to examine all consistent cuts of the computation, twenty eight in total, to ascertain whether some consistent cut satisfies the predicate. Alternatively, we can compute a slice of the computation with respect to the predicate  $(x_1 \geq 1) \wedge (x_3 \leq 3)$  as portrayed in Fig. 1(b). The slice is modeled by a directed graph. Each vertex of the graph corresponds to a subset of events. If a vertex is contained in a consistent cut, the interpretation is that all events corresponding to the vertex are contained in the cut. Moreover, a vertex belongs to a consistent cut only if all its incoming neighbours are also present in the cut. We can now restrict our search to the consistent cuts of the slice which are only six in number, namely  $\{a, e, f, u, v\}$ ,  $\{a, e, f, u, v, b\}$ ,  $\{a, e, f, u, v, w\}$ ,  $\{a, e, f, u, v, b, w\}$ ,  $\{a, e, f, u, v, w, g\}$  and  $\{a, e, f, u, v, b, w, g\}$ . The slice has much fewer consistent cuts than the computation itself—exponentially smaller in many cases—resulting in substantial savings.

We also show that the results pertaining to consistent global checkpoints [14, 18] can be derived as special cases of computation slicing. In particular, we furnish an alternate characterization of the condition under which individual local checkpoints can be combined with others to form a consistent global checkpoint (consistency theorem by Netzer and Xu [14]): a set of local checkpoints can belong to the same consistent global snapshot iff the local checkpoints in the set are mutually consistent (including with itself) in the slice. Moreover, the R-graph (rollback-dependency graph) defined by Wang [18] is a special case of the slice. The minimum and maximum consistent global checkpoints that contain a set of local checkpoints [18] can also be easily obtained using the slice.

In summary, this paper makes the following contributions:

- In Section 3, we generalize the notion of computation slice introduced in our earlier paper [6]. We show that slice exists for all global predicates in Section 4
- We establish that it is, in general, NP-complete to determine whether a global predicate has a non-empty slice in Section 4.
- In Section 4, an application of computation slicing to monitoring global properties in distributed systems is provided. Specifically, we give an algorithm to determine whether a global predicate satisfying certain properties is possibly true, invariant or controllable in a distributed computation using slicing.
- We present an efficient representation of slice in Section 5 that we use later to devise an efficient algorithm to *graft* two slices in Section 6. Grafting can be done in two ways. Given two slices, we can either compute the smallest slice that contains all consistent cuts that are common to both slices or compute the smallest slice that contains all consistent cuts that belong to at least one of the slices. An efficient algorithm using grafting to compute slice for complement of a regular predicate, called *co-regular* predicate, is provided. We also show how grafting can be used to avoid examining many consistent cuts when detecting a predicate.
- We provide an optimal algorithm to compute slices for special cases of regular predicates in Section 7. In our earlier paper [6], the algorithm to compute slices has  $O(N^2|E|)$  time complexity, where  $N$  is the number of processes

and  $E$  is the set of events in the distributed system. The algorithm presented in this paper has  $O(|E|)$  complexity which is optimal.

- Finally, in Section 7, we show that the results pertaining to consistent global checkpoints [14, 18] can be derived as special cases of computation slicing.

Due to lack of space, the proofs of lemmas, theorems and corollaries, and other details have been omitted. Interested reader can find them in the technical report [12].

## 2 Model and Notation

### 2.1 Lattices

Given a lattice, we use  $\sqcap$  and  $\sqcup$  to denote its meet (infimum) and join (supremum) operators, respectively. A lattice is *distributive* iff meet distributes over join. Formally,  $a \sqcap (b \sqcup c) \equiv (a \sqcap b) \sqcup (a \sqcap c)$ .

### 2.2 Directed Graphs: Path- and Cut-Equivalence

Traditionally, a distributed computation is modeled by a partial order on a set of events. We use directed graphs to model both distributed computation and slice. Directed graphs allow us to handle both of them in a convenient and uniform manner.

Given a directed graph  $G$ , let  $V(G)$  and  $E(G)$  denote its set of vertices and edges, respectively. A subset of vertices of a directed graph form a *consistent cut* iff the subset contains a vertex only if it contains all its incoming neighbours. Formally,

$$C \text{ is a consistent cut of } G \triangleq \langle \forall e, f \in V(G) : (e, f) \in E(G) : f \in C \Rightarrow e \in C \rangle$$

Observe that a consistent cut either contains all vertices in a cycle or none of them. This observation can be generalized to a strongly connected component. Traditionally, the notion of consistent cut (*down-set* or *order ideal*) is defined for partially ordered sets [5]. Here, we extend the notion to sets with arbitrary orders. Let  $\mathcal{C}(G)$  denote the set of consistent cuts of a directed graph  $G$ . Observe that the empty set  $\emptyset$  and the set of vertices  $V(G)$  trivially belong to  $\mathcal{C}(G)$ . We call them *trivial* consistent cuts. The following theorem is a slight generalization of the result in lattice theory that the set of down-sets of a partially ordered set forms a distributive lattice [5].

**Theorem 1.** *Given a directed graph  $G$ ,  $\langle \mathcal{C}(G); \subseteq \rangle$  forms a distributive lattice.*

The theorem follows from the fact that, given two consistent cuts of a graph, the cuts given by their set intersection and set union are also consistent.

A directed graph  $G$  is *cut-equivalent* to a directed graph  $H$  iff they have the same set of consistent cuts, that is,  $\mathcal{C}(G) = \mathcal{C}(H)$ . Let  $\mathcal{P}(G)$  denote the set of pairs of vertices  $(u, v)$  such that there is a path from  $u$  to  $v$  in  $G$ . We assume

that each vertex has a path to itself. A directed graph  $G$  is *path-equivalent* to a directed graph  $H$  iff a path from vertex  $u$  to vertex  $v$  in  $G$  implies a path from vertex  $u$  to vertex  $v$  in  $H$  and vice versa, that is,  $\mathcal{P}(G) = \mathcal{P}(H)$ .

**Lemma 1.** *Let  $G$  and  $H$  be directed graphs on the same set of vertices. Then,*

$$\mathcal{P}(G) \subseteq \mathcal{P}(H) \equiv \mathcal{C}(G) \supseteq \mathcal{C}(H)$$

Lemma 1 implies that two directed graphs are cut-equivalent iff they are path-equivalent. This is significant because path-equivalence can be verified in polynomial-time ( $|\mathcal{P}(G)| = O(|V(G)|^2)$ ) as compared to cut-equivalence which is computationally expensive to ascertain in general ( $|\mathcal{C}(G)| = O(2^{|V(G)|})$ ).

### 2.3 Distributed Computations as Directed Graphs

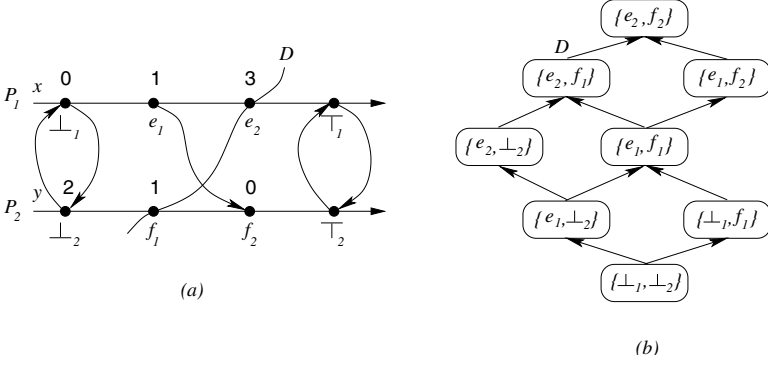
We assume an asynchronous distributed system [12] with the set of processes  $P = \{P_1, P_2, \dots, P_N\}$ . Processes communicate and synchronize with each other by sending messages over a set of reliable channels.

A *local computation* of a process is described by a sequence of events that transforms the *initial state* of the process into the *final state*. At each step, the *local state* of a process is captured by the initial state and the sequence of events that have been executed up to that step. Each event is a *send event*, a *receive event*, or an *internal event*. An event causes the local state of a process to be updated. Additionally, a send event causes a message to be sent and a receive event causes a message to be received. We assume the presence of fictitious *initial* and *final events* on each process  $P_i$ , denoted by  $\perp_i$  and  $\top_i$ , respectively. The initial event occurs before any other event on the process and initializes the state of the process. The final event occurs after all other events on the process.

Let  $proc(e)$  denote the process on which event  $e$  occurs. The predecessor and successor events of  $e$  on  $proc(e)$  are denoted by  $pred(e)$  and  $succ(e)$ , respectively, if they exist. We denote the order of events on process  $P_i$  by  $\sim_{P_i}$ . Let  $\sim_P$  be the union of all  $\sim_{P_i}$ s,  $1 \leq i \leq N$ , and  $\simeq_P$  denote the reflexive closure of  $\sim_P$ .

We model a *distributed computation* (or simply a *computation*), denoted by  $\langle E, \rightarrow \rangle$ , as a directed graph with vertices as the set of events  $E$  and edges as  $\rightarrow$ . To limit our attention to only those consistent cuts that can actually occur during an execution, we assume that, for any computation  $\langle E, \rightarrow \rangle$ ,  $\mathcal{P}(\langle E, \rightarrow \rangle)$  contains at least the Lamport's happened-before relation [10]. We assume that the set of all initial events belong to the same strongly connected component. Similarly, the set of all final events belong to the same strongly connected component. This ensures that any non-trivial consistent cut will contain all initial events and none of the final events. As a result, every consistent cut of a computation in traditional model is a non-trivial consistent cut of the computation in our model and vice versa. Only non-trivial consistent cuts are of real interest to us. We will see later that our model allows us to capture empty slices in a very convenient fashion.

A distributed computation in our model can contain cycles. This is because whereas a computation in the happened-before model captures the observable



**Fig. 2.** (a) A computation and (b) the lattice corresponding to its consistent cuts.

order of execution of events, a computation in our model captures the set of possible consistent cuts.

A *frontier* of a consistent cut is the set of those events of the cut whose successors, if they exist, are not contained in the cut. Formally,

$$\text{frontier}(C) \triangleq \{e \in C \mid \text{succ}(e) \text{ exists} \Rightarrow \text{succ}(e) \notin C\}$$

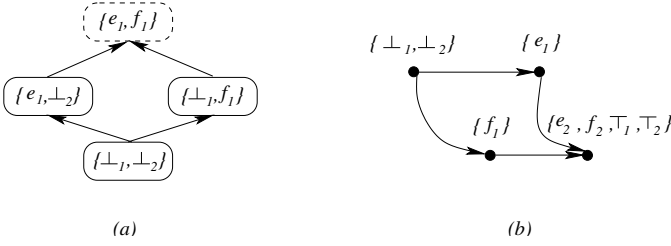
A consistent cut is uniquely characterized by its frontier and vice versa. Thus sometimes, especially in figures, we specify a consistent cut by simply listing the events in its frontier instead of enumerating all its events. Two events are said to be *consistent* iff they are contained in the frontier of some consistent cut, otherwise they are *inconsistent*. It can be verified that events  $e$  and  $f$  are consistent iff there is no path in the computation from  $\text{succ}(e)$ , if it exists, to  $f$  and from  $\text{succ}(f)$ , if it exists, to  $e$ . Also, note that, in our model, an event can be inconsistent with itself. Fig. 2 depicts a computation and the lattice of its (non-trivial) consistent cuts. A consistent cut in the figure is represented by its frontier. For example, the consistent cut  $D$  is represented by  $\{e_2, f_1\}$ .

## 2.4 Global Predicates

A *global predicate* (or simply a *predicate*) is a boolean-valued function defined on variables of processes. It is evaluated on events in the frontier of a consistent cut. Some examples are mutual exclusion and “at least one philosopher does not have any fork”. We leave the predicate undefined for the trivial consistent cuts. A global predicate is *local* iff it depends on variables of at most one process. For example, “ $P_i$  is in red state” and “ $P_i$  does not have the token”.

## 3 Slicing a Distributed Computation

In this section, we define the notion of slice of a computation with respect to a predicate. The definition given here is weaker than the definition given in our



**Fig. 3.** (a) The sublattice of the lattice in Fig. 2(b) with respect to the predicate  $((x < 2) \wedge (y > 1)) \vee (x < 1)$ , and (b) the corresponding slice.

earlier paper [6]. However, slice now exists with respect to every predicate (not just specific predicates).

**Definition 1 (Slice).** *A slice of a computation with respect to a predicate is the smallest directed graph (with minimum number of consistent cuts) that contains all consistent cuts of the original computation that satisfy the predicate.*

We will later show that the smallest computation is well-defined for every predicate. A slice of computation  $\langle E, \rightarrow \rangle$  with respect to a predicate  $b$  is denoted by  $\langle E, \rightarrow \rangle_b$ . Note that  $\langle E, \rightarrow \rangle = \langle E, \rightarrow \rangle_{\text{true}}$ . In the rest of the paper, we use the terms “computation”, “slice” and “directed graph” interchangeably.

Fig. 3(a) depicts the set of consistent cuts of the computation in Fig. 2(a) that satisfy the predicate  $((x < 2) \wedge (y > 1)) \vee (x < 1)$ . The cut shown with dashed outline does not actually satisfy the predicate but has to be included to complete the sublattice. Fig. 3(b) depicts the slice of the computation with respect to the predicate. In the figure, all events in a subset belong to the same strongly connected component.

In our model, every slice derived from the computation  $\langle E, \rightarrow \rangle$  will have the trivial consistent cuts ( $\emptyset$  and  $E$ ) among its set of consistent cuts. Consequently, a slice is *empty* iff it has no non-trivial consistent cuts. In the rest of the paper, unless otherwise stated, a consistent cut refers to a non-trivial consistent cut.

A slice of a computation with respect to a predicate is *lean* iff every consistent cut of the slice satisfies the predicate.

## 4 Regular Predicates

A global predicate is *regular* iff the set of consistent cuts that satisfy the predicate forms a *sublattice* of the lattice of consistent cuts [6]. Equivalently, if two consistent cuts satisfy a regular predicate then the cuts given by their set intersection and set union will also satisfy the predicate. Some examples of regular predicates are any local predicate and channel predicates such as “there are at most  $k$  messages in transit from  $P_i$  to  $P_j$ ”. The class of regular predicates is closed under conjunction [6]. We prove elsewhere [6] that the slice of a computation with respect to a predicate is lean iff the predicate is regular. We next show how

slicing can be used to monitor predicates in distributed systems. Later, we use the notion of regular predicates to prove that the slice exists and is well-defined with respect to every predicate.

#### 4.1 Using Slices to Monitor Regular Predicates

A predicate can be monitored under four modalities, namely *possibly*, *definitely*, *invariant* and *controllable* [3,7,17,11]. A predicate is *possibly* true in a computation iff there is a consistent cut of the computation that satisfies the predicate. On the other hand, a predicate *definitely* holds in a computation iff it eventually becomes true in all runs of the computation (a *run* is a path in the lattice of consistent cuts). The predicates *invariant*: $b$  and *controllable*: $b$  are duals of predicates *possibly*: $b$  and *controllable*: $b$ , respectively. Predicate detection normally involves detecting a predicate under *possibly* modality whereas predicate control involves monitoring a predicate under *controllable* modality. Monitoring has applications in the areas of testing and debugging and software fault-tolerance of distributed programs.

The next theorem describes how *possibly*: $b$ , *invariant*: $b$  and *controllable*: $b$  can be computed using the notion of slice when  $b$  is a regular predicate. We do not yet know the complexity of computing *definitely*: $b$  when  $b$  is regular.

**Theorem 2.** *A regular predicate is*

1. **possibly** true in a computation iff the slice of the computation with respect to the predicate has at least one non-trivial consistent cut, that is, it has at least two strongly connected components.
2. **invariant** in a computation iff the slice of the computation with respect to the predicate is cut-equivalent to the computation.
3. **controllable** in a computation iff the slice of the computation with respect to the predicate has the same number of strongly connected components as the computation.

Observe that the first proposition holds for any arbitrary predicate. Since detecting whether a predicate possibly holds in a computation is NP-complete in general [2,15,13], it is, in general, NP-complete to determine whether a predicate has a non-empty slice.

#### 4.2 Regularizing a Non-regular Predicate

In this section, we show that slice exists and is well-defined with respect to every predicate. We know that it is true for at least regular predicates [6]. In addition, the slice with respect to a regular predicate is lean. We exploit these facts and define a closure operator, denoted by *reg*, which, given a computation, converts an arbitrary predicate into a regular predicate satisfying certain properties. Given a computation, let  $\mathcal{R}$  denote the set of predicates that are regular with respect to the computation.

**Definition 2 (reg).** Given a predicate  $b$ , we define  $\text{reg}(b)$  as the predicate that satisfies the following conditions:

1. it is regular, that is,  $\text{reg}(b) \in \mathcal{R}$ ,
2. it is weaker than  $b$ , that is,  $b \Rightarrow \text{reg}(b)$ , and
3. it is stronger than any other predicate that satisfies 1 and 2, that is,  $\langle \forall b' : b' \in \mathcal{R} : (b \Rightarrow b') \Rightarrow (\text{reg}(b) \Rightarrow b') \rangle$

Informally,  $\text{reg}(b)$  is the *strongest regular predicate weaker than  $b$* . In general,  $\text{reg}(b)$  not only depends on the predicate  $b$  but also on the computation under consideration. We assume the dependence on computation to be implicit and make it explicit only when necessary. The next theorem establishes that  $\text{reg}(b)$  exists for every predicate. Observe that the slice for  $b$  is given by the slice for  $\text{reg}(b)$ . Thus slice exists and is well-defined for all predicates.

**Theorem 3.** Given a predicate  $b$ ,  $\text{reg}(b)$  exists and is well-defined.

Thus, given a computation  $\langle E, \rightarrow \rangle$  and a predicate  $b$ , the slice of  $\langle E, \rightarrow \rangle$  with respect to  $b$  can be obtained by first applying  $\text{reg}$  operator to  $b$  to get  $\text{reg}(b)$  and then computing the slice of  $\langle E, \rightarrow \rangle$  with respect to  $\text{reg}(b)$ .

**Theorem 4.**  $\text{reg}$  is a closure operator. Formally,

1.  $\text{reg}(b)$  is weaker than  $b$ , that is,  $b \Rightarrow \text{reg}(b)$ ,
2.  $\text{reg}$  is monotonic, that is,  $(b \Rightarrow b') \Rightarrow (\text{reg}(b) \Rightarrow \text{reg}(b'))$ , and
3.  $\text{reg}$  is idempotent, that is,  $\text{reg}(\text{reg}(b)) \equiv \text{reg}(b)$ .

From the above theorem it follows that [5, Theorem 2.21],

**Corollary 1.**  $\langle \mathcal{R}; \Rightarrow \rangle$  forms a lattice.

The meet and join of two regular predicates  $b_1$  and  $b_2$  is given by

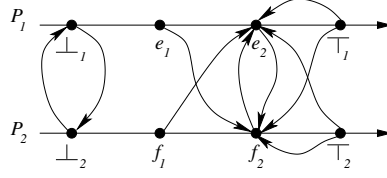
$$\begin{aligned} b_1 \sqcap b_2 &\triangleq b_1 \wedge b_2 \\ b_1 \sqcup b_2 &\triangleq \text{reg}(b_1 \vee b_2) \end{aligned}$$

The dual notion of  $\text{reg}(b)$ , the weakest regular predicate stronger than  $b$ , is conceivable. However, such a predicate may not always be unique [12].

## 5 Representing a Slice

Observe that any directed graph that is cut-equivalent or path-equivalent to a slice constitutes its valid representation. However, for computational purposes, it is preferable to select those graphs to represent a slice that have fewer edges and can be constructed cheaply. In this section, we show that every slice can be represented by a directed graph with  $O(|E|)$  vertices and  $O(N|E|)$  edges. Furthermore, the graph can be built in  $O(N^2|E|)$  time.





**Fig. 4.** The skeletal representation of the slice in Fig. 3(b) (without self-loops).

Given a computation  $\langle E, \rightarrow \rangle$ , a regular predicate  $b$  and an event  $e$ , let  $J_b(e)$  denote the least consistent cut of  $\langle E, \rightarrow \rangle$  that contains  $e$  and satisfies  $b$ . If  $J_b(e)$  does not exist then it is set to the trivial consistent cut  $E$ . Here, we use  $E$  as a *sentinel* cut. Fig. 4 depicts a directed graph that represents the slice shown in Fig. 3(b). In the figure,  $J_b(e_1) = \{\perp_1, e_1, \perp_2\}$  and  $J_b(f_2) = \{\perp_1, e_1, e_2, \top_1, \perp_2, f_1, f_2, \top_2\}$ .

The cut  $J_b(e)$  can also be viewed as the least consistent cut of the slice  $\langle E, \rightarrow \rangle_b$  that contains the event  $e$ . The results in [6] establish that it is sufficient to know  $J_b(e)$  for each event  $e$  in order to recover the slice. In particular, a directed graph with  $E$  as the set of vertices and an edge from an event  $e$  to an event  $f$  iff  $J_b(e) \subseteq J_b(f)$  is cut-equivalent to the slice  $\langle E, \rightarrow \rangle_b$ . We also present an  $O(N^2|E|)$  algorithm to compute  $J_b(e)$  for each event  $e$ . However, the graph so obtained can have as many as  $\Omega(|E|^2)$  edges.

Let  $F_b(e, i)$  denote the earliest event  $f$  on  $P_i$  such that  $J_b(e) \subseteq J_b(f)$ . Informally,  $F_b(e, i)$  is the earliest event on  $P_i$  that is reachable from  $e$  in the slice  $\langle E, \rightarrow \rangle_b$ . For example, in Fig. 4,  $F_b(e_1, 1) = e_1$  and  $F_b(e_1, 2) = f_2$ . Given  $J_b(e)$  for each event  $e$ ,  $F_b(e, i)$  for each event  $e$  and process  $P_i$  can be computed in  $O(N|E|)$  time [12]. We now construct a directed graph that we call the *skeletal representation* of the slice with respect to  $b$  and denote it by  $G_b$ . The graph  $G_b$  has  $E$  as the set of vertices and the following edges: (1) for each event  $e$ , that is not a final event, there is an edge from  $e$  to  $\text{succ}(e)$ , and (2) for each event  $e$  and process  $P_i$ , there is an edge from  $e$  to  $F_b(e, i)$ .

The skeletal representation of the slice depicted in Fig. 3(b) is shown in Fig. 4. To prove that the graph  $G_b$  is actually cut-equivalent to the slice  $\langle E, \rightarrow \rangle_b$ , it suffices to show the following:

**Theorem 5.** For events  $e$  and  $f$ ,  $J_b(e) \subseteq J_b(f) \equiv (e, f) \in \mathcal{P}(G_b)$ .

Besides having computational benefits, the skeletal representation of a slice can be used to devise a simple and efficient algorithm to graft two slices.

## 6 Grafting Two Slices

In this section, we present algorithm to graft two slices which can be done with respect to meet or join. Informally, the former case corresponds to the smallest slice that contains all consistent cuts common to both slices whereas the latter case corresponds to the smallest slice that contains consistent cuts of both slices.

In other words, given slices  $\langle E, \rightarrow \rangle_{b_1}$  and  $\langle E, \rightarrow \rangle_{b_2}$ , where  $b_1$  and  $b_2$  are regular predicates, we provide algorithm to compute the slice  $\langle E, \rightarrow \rangle_b$ , where  $b$  is either  $b_1 \sqcap b_2 = b_1 \wedge b_2$  or  $b_1 \sqcup b_2 = \text{reg}(b_1 \vee b_2)$ . Grafting enables us to compute the slice for an arbitrary boolean expression of local predicates—by rewriting it in DNF—although it may require exponential time in the worst case. Later, in this section, we present an efficient algorithm based on grafting to compute slice for a *co-regular* predicate (complement of a regular predicate). We also show how grafting can be used to avoid examining many consistent cuts when detecting a predicate under *possibly* modality.

### 6.1 Grafting with Respect to Meet: $\mathbf{b} \equiv \mathbf{b}_1 \sqcap \mathbf{b}_2 \equiv \mathbf{b}_1 \wedge \mathbf{b}_2$

In this case, the slice  $\langle E, \rightarrow \rangle_b$  contains a consistent cut of  $\langle E, \rightarrow \rangle$  iff the cut satisfies  $b_1$  as well as  $b_2$ . Let  $F_{\min}(e, i)$  denote the earlier of events  $F_{b_1}(e, i)$  and  $F_{b_2}(e, i)$ , that is,  $F_{\min}(e, i) = \min\{F_{b_1}(e, i), F_{b_2}(e, i)\}$ . The following lemma establishes that, for each event  $e$  and process  $P_i$ ,  $F_{\min}(e, i)$  cannot occur before  $F_b(e, i)$ .

**Lemma 2.** *For each event  $e$  and process  $P_i$ ,  $F_b(e, i) \simeq_P F_{\min}(e, i)$ .*

We now construct a directed graph  $G_{\min}$  which is similar to  $G_b$ , the skeletal representation for  $\langle E, \rightarrow \rangle_b$ , except that we use  $F_{\min}(e, i)$  instead of  $F_b(e, i)$  in its construction. The next theorem proves that  $G_{\min}$  is cut-equivalent to  $G_b$ .

**Theorem 6.**  *$G_{\min}$  is cut-equivalent to  $G_b$ .*

Roughly speaking, the aforementioned algorithm computes the union of the sets of edges of each slice. Note that, in general,  $F_b(e, i)$  need not be same as  $F_{\min}(e, i)$  [12]. This algorithm can be generalized to conjunction of an arbitrary number of regular predicates.

### 6.2 Grafting with Respect to Join: $\mathbf{b} \equiv \mathbf{b}_1 \sqcup \mathbf{b}_2 \equiv \text{reg}(\mathbf{b}_1 \vee \mathbf{b}_2)$

In this case, the slice  $\langle E, \rightarrow \rangle_b$  contains a consistent cut of  $\langle E, \rightarrow \rangle$  if the cut satisfies either  $b_1$  or  $b_2$ . The dual of the graph  $G_{\min}$ —min replaced by max—denoted by  $G_{\max}$  (surprisingly) turns out to be cut-equivalent to the slice  $\langle E, \rightarrow \rangle_b$ . As before, let  $F_{\max}(e, i)$  denote the later of events  $F_{b_1}(e, i)$  and  $F_{b_2}(e, i)$ , that is,  $F_{\max}(e, i) = \max\{F_{b_1}(e, i), F_{b_2}(e, i)\}$ . The following lemma establishes that, for each event  $e$  and process  $P_i$ ,  $F_b(e, i)$  cannot occur before  $F_{\max}(e, i)$ .

**Lemma 3.** *For each event  $e$  and process  $P_i$ ,  $F_{\max}(e, i) \simeq_P F_b(e, i)$ .*

We now construct a directed graph  $G_{\max}$  that is similar to  $G_b$ , the skeletal representation for  $\langle E, \rightarrow \rangle_b$ , except that we use  $F_{\max}(e, i)$  instead of  $F_b(e, i)$  in its construction. The next theorem proves that  $G_{\max}$  is cut-equivalent to  $G_b$ .

**Theorem 7.**  *$G_{\max}$  is cut-equivalent to  $G_b$ .*

Intuitively, the above-mentioned algorithm computes the intersection of the sets of edges of each slice. In this case, in contrast to the former case,  $F_b(e, i)$  is actually identical to  $F_{\max}(e, i)$  [12]. This algorithm can be generalized to disjunction of an arbitrary number of regular predicates.

### 6.3 Applications of Grafting

**Computing Slice for a Co-Regular Predicate.** Given a regular predicate, we give an algorithm to compute the slice of a computation with respect to its negation—a co-regular predicate. In particular, we express the negation as disjunction of polynomial number of regular predicates. The slice can then be computed by grafting together slices for each disjunct.

Let  $\langle E, \rightarrow \rangle$  be a computation and  $\langle E, \rightarrow \rangle_b$  be its slice with respect to a regular predicate  $b$ . For convenience, let  $\rightarrow_b$  be the edge relation for the slice. *We assume that both  $\rightarrow$  and  $\rightarrow_b$  are transitive relations.* Our objective is to find a property that *distinguishes* the consistent cuts that belong to the slice from the consistent cuts that do not. Consider events  $e$  and  $f$  such that  $e \not\rightarrow f$  but  $e \rightarrow_b f$ . Then, clearly, a consistent cut that contains  $f$  but does not contain  $e$  cannot belong to the slice. On the other hand, every consistent cut of the slice that contains  $f$  also contains  $e$ . This motivates us to define a predicate  $prevents(f, e)$  as follows:

$$C \text{ satisfies } prevents(f, e) \triangleq (f \in C) \wedge (e \notin C)$$

It can be proved that  $prevents(f, e)$  is a regular predicate [12]. It turns out that every consistent cut that does not belong to the slice satisfies  $prevents(f, e)$  for some events  $e$  and  $f$  such that  $(e \not\rightarrow f) \wedge (e \rightarrow_b f)$  holds. Formally,

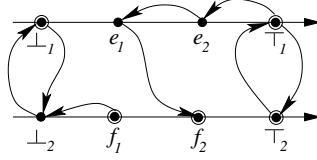
**Theorem 8.** *Let  $C$  be a consistent cut of  $\langle E, \rightarrow \rangle$ . Then,*

$$C \text{ satisfies } \neg b \equiv \langle \exists e, f : (e \rightarrow_b f) \wedge (e \not\rightarrow f) : C \text{ satisfies } prevents(f, e) \rangle$$

Theorem 8 implies that  $\neg b$  can be expressed as disjunction of  $prevents$ 's.

**Pruning State Space for Predicate Detection.** Detecting a predicate under *possibly* modality is NP-complete in general [2, 15, 13]. Using grafting, we can reduce the search space for predicates composed from local predicates using  $\neg$ ,  $\wedge$  and  $\vee$  operators. We first transform the predicate into an equivalent predicate in which  $\neg$  is applied directly to the local predicates and never to more complex expressions. Observe that the negation of a local predicate is also a local predicate. We start by computing slices with respect to these local predicates. This can be done because a local predicate is regular and hence the algorithm given in [6] can be used to compute the slice. We then recursively graft slices together, with respect to the appropriate operator, working our way out from the local predicates until we reach the whole predicate. This will give us a slice of the computation—not necessarily the smallest—which contains all consistent cuts of the computation that satisfy the predicate. In many cases, the slice obtained will be much smaller than the computation itself enabling us to ignore many consistent cuts in our search.

For example, suppose we wish to compute the slice of a computation with respect to the predicate  $(x_1 \vee x_2) \wedge (x_3 \vee x_4)$ , where  $x_i$  is a boolean variable on process  $p_i$ . As explained, we first compute slices for the local predicates  $x_1$ ,  $x_2$ ,  $x_3$  and  $x_4$ . We then graft the first two and the last two slices together with



**Fig. 5.** An optimal algorithm to compute the slice for a conjunctive predicate.

respect to join to obtain slices for the clauses  $x_1 \vee x_2$  and  $x_3 \vee x_4$ , respectively. Finally, we graft the slices for both clauses together with respect to meet to get the slice for the predicate  $reg(x_1 \vee x_2) \wedge reg(x_3 \vee x_4)$  which, in general, is larger than the slice for the predicate  $(x_1 \vee x_2) \wedge (x_3 \vee x_4)$  but much smaller than the computation itself.

The result of Section 6.3 allows us to generalize this approach to predicates composed from arbitrary regular predicates using  $\neg$ ,  $\wedge$  and  $\vee$  operators. We plan to conduct experiments to quantitatively evaluate the effectiveness of our approach. Although our focus is on detecting predicates under *possibly* modality, slicing can be used to prune search space for monitoring predicates under other modalities too.

## 7 Optimal Algorithm for Slicing

The algorithm we presented in 6 to compute slices for regular predicates has  $O(N^2|E|)$  time complexity, where  $N$  is the number of processes and  $E$  is the set of events. In this section we present an *optimal* algorithm for computing slices for special cases of regular predicates. Our algorithm will have  $O(|E|)$  time complexity. Due to lack of space, only the optimal algorithm for conjunctive predicates is presented. The optimal algorithm for other regular predicates such as channel predicates can be found elsewhere [12].

A *conjunctive predicate* is a conjunction of local predicates. For example, “ $P_1$  is in red state”  $\wedge$  “ $P_2$  is in green state”  $\wedge$  “ $P_3$  is in blue state”. Given a set of local predicates, one for each process, we can categorize events on each process into *true events* and *false events*. An event is a true event iff the corresponding local predicate evaluates to true, otherwise it is a false event.

To compute the slice of a computation for a conjunctive predicate, we construct a directed graph with vertices as events in the computation and the following edges: (1) from an event, that is not a final event, to its successor, (2) from a send event to the corresponding receive event, and (3) from the successor of a false event to the false event.

For the purpose of building the graph, we assume that all final events are true events. Thus every false event has a successor. The first two kinds of edges ensure that the Lamport’s happened-before relation is captured in the graph. The algorithm is illustrated Fig. 5. In the figure, all true events have been encircled.

It can be proved that the directed graph obtained is cut-equivalent to the slice of the computation with respect to the given conjunctive predicate [4]. It is easy

to see that the graph has  $O(|E|)$  vertices,  $O(|E|)$  edges (at most three edges per event assuming that an event that is not local either sends at most one message or receives at most one message but not both) and can be built in  $O(|E|)$  time. The slice can be computed by finding out the strongly connected components of the graph [4]. Thus the algorithm has  $O(|E|)$  overall time complexity. It also gives us an  $O(|E|)$  algorithm to evaluate *possibly: b* when *b* is a conjunctive predicate (see Theorem 2).

By defining a local predicate (evaluated on an event) to be true iff the event corresponds to a local checkpoint, it can be verified that there is a *zigzag path* [14,18] from a local checkpoint *c* to a local checkpoint *c'* in a computation iff there is a path from *succ(c)*, if it exists, to *c'* in the corresponding slice—which can be ascertained by comparing  $J_b(\text{succ}(c))$  and  $J_b(c')$ . An alternative formulation of the consistency theorem in [14] can thus be obtained as follows:

**Theorem 9.** *A set of local checkpoints can belong to the same consistent global snapshot iff the local checkpoints in the set are mutually consistent (including with itself) in the corresponding slice.*

Moreover, the R-graph (rollback-dependency graph) [18] is path-equivalent to the slice when each contiguous sequence of false events on a process is merged with the nearest true event that occurs later on the process. The minimum consistent global checkpoint that contains a set of local checkpoints [18] can be computed by taking the set union of  $J_b$ 's for each local checkpoint in the set. The maximum consistent global checkpoint can be similarly obtained by using the dual of  $J_b$ .

## 8 Conclusion and Future Work

In this paper, the notion of slice introduced in our earlier paper [6] is generalized and its existence for all global predicates is established. The intractability of computing the slice, in general, is also proved. An optimal algorithm to compute slices for special cases of predicates is provided. Moreover, an efficient algorithm to graft two slices is also given. Application of slicing in general and grafting in particular to global property evaluation of distributed programs is discussed. Finally, the results pertaining to consistent global checkpoints [14,18] are shown to be special cases of computation slicing.

As future work, we plan to study grafting in greater detail. Specifically, we plan to conduct experiments to quantitatively evaluate its effectiveness in weeding out unnecessary consistent cuts from examination during state space search for predicate detection. Another direction for future research is to extend the notion of slicing to include temporal predicates.

## References

1. K. M. Chandy and L. Lamport. Distributed Snapshots: Determining Global States of Distributed Systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

2. C. Chase and V. K. Garg. Detection of Global Predicates: Techniques and their Limitations. *Distributed Computing*, 11(4):191–201, 1998.
3. R. Cooper and K. Marzullo. Consistent Detection of Global Predicates. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 163–173, Santa Cruz, California, 1991.
4. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, 1990.
5. B. A. Davey and H. A. Priestley. *Introduction to Lattices and Order*. Cambridge University Press, Cambridge, UK, 1990.
6. V. K. Garg and N. Mittal. On Slicing a Distributed Computation. In *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 322–329, Phoenix, Arizona, April 2001.
7. V. K. Garg and B. Waldecker. Detection of Unstable Predicates. In *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, Santa Cruz, California, May 1991.
8. B. Korel and R. Ferguson. Dynamic Slicing of Distributed Programs. *Applied Mathematics and Computer Science Journal*, 2(2):199–215, 1992.
9. B. Korel and J. Rilling. Application of Dynamic Slicing in Program Debugging. In Mariam Kamkar, editor, *Proceedings of the 3rd International Workshop on Automated Debugging (AADEBUG)*, pages 43–57, Linköping, Sweden, May 1997.
10. L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM (CACM)*, 21(7):558–565, July 1978.
11. N. Mittal and V. K. Garg. Debugging Distributed Programs Using Controlled Re-execution. In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing (PODC)*, pages 239–248, Portland, Oregon, July 2000.
12. N. Mittal and V. K. Garg. Computation Slicing: Techniques and Theory. Technical Report TR-PDS-2001-002, The Parallel and Distributed Systems Laboratory, Department of Electrical and Computer Engineering, The University of Texas at Austin, April 2001. Available at <http://www.cs.utexas.edu/users/neerajm>.
13. N. Mittal and V. K. Garg. On Detecting Global Predicates in Distributed Computations. In *Proceedings of the 21st IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 3–10, Phoenix, Arizona, April 2001.
14. R. H. B. Netzer and J. Xu. Necessary and Sufficient Conditions for Consistent Global Snapshots. *IEEE Transactions on Parallel and Distributed Systems*, 6(2):165–169, February 1995.
15. S. D. Stoller and F. Schnieder. Faster Possibility Detection by Combining Two Approaches. In *Proceedings of the Workshop on Distributed Algorithms (WDAG)*, France, September 1995.
16. A. Tarafdar and V. K. Garg. Predicate Control for Active Debugging of Distributed Programs. In *Proceedings of the 9th IEEE Symposium on Parallel and Distributed Processing (SPDP)*, Orlando, 1998.
17. A. Tarafdar and V. K. Garg. Software Fault Tolerance of Concurrent Programs Using Controlled Re-execution. In *Proceedings of the 13th Symposium on Distributed Computing (DISC)*, pages 210–224, Bratislava, Slovak Republic, September 1999.
18. Yi-Min Wang. Consistent Global Checkpoints that Contain a Given Set of Local Checkpoints. *IEEE Transactions on Computers*, 46(4):456–468, April 1997.
19. M. Weiser. Programmers Use Slices when Debugging. *Communications of the ACM (CACM)*, 25(7):446–452, 1982.

# A Low-Latency Non-blocking Commit Service<sup>\*</sup>

R. Jiménez-Peris<sup>1</sup>, M. Patiño-Martínez<sup>1</sup>, G. Alonso<sup>2</sup>, and S. Arévalo<sup>3</sup>

<sup>1</sup> Technical University of Madrid (UPM), Facultad de Informática,  
E-28660 Boadilla del Monte, Madrid, Spain, {rjimenez, mpatino}@fi.upm.es

<sup>2</sup> Swiss Federal Institute of Technology (ETHZ), Department of Computer Science,  
CLU E1, ETH-Zentrum, CH-8092 Zürich, Switzerland, alonso@inf.ethz.ch

<sup>3</sup> Universidad Rey Juan Carlos, Escuela de Ciencias Experimentales, Móstoles,  
Madrid, Spain, s.arevalo@escet.urjc.es

**Abstract.** Atomic commitment is one of the key functionalities of modern information systems. Conventional distributed databases, transaction processing monitors, or distributed object platforms are examples of complex systems built around atomic commitment. The vast majority of such products implement atomic commitment using some variation of 2 Phase Commit (2PC) although 2PC may block under certain conditions. The alternative would be to use non-blocking protocols but these are seen as too heavy and slow. In this paper we propose a non-blocking distributed commit protocol that exhibits the same latency as 2PC. The protocol combines several ideas (optimism and replication) to implement a scalable solution that can be used in a wide range of applications.

## 1 Introduction

Atomic commitment (AC) protocols are used to implement atomic transactions. *Two-phase commit* (2PC) [Gra78] is the most widely used AC protocol although its blocking behavior is well known. There are also non-blocking protocols but they have an inherent higher cost [DS83,KR01] usually translated in either an explicit extra round of messages (*3 phase commit* (3PC) [Ske81,Ske82,KD95]) or an implicit one (when using uniform multicast [BT93]).

The reason why 2PC is the standard protocol for atomic commitment is that transactional systems pay as much attention to performance as they do to consistency. For instance, most systems summarily abort those transactions that have not committed after a given period of time so that they do not keep resources locked. Existing non-blocking protocols resolve the consistency problem by increasing the latency and, therefore, are not practical. A realistic non-blocking alternative to 2PC needs to consider both consistency and transaction latency. Ideally, the non-blocking protocol should have the same latency as 2PC. Our goal is to implement such a protocol by addressing the three main sources of delay in atomic commitment: message overhead, forced writes to the log, and the *convoy effect* caused by transactions waiting for other transactions to commit.

---

<sup>\*</sup> This research has been partially funded by the Spanish National Research Council CICYT under grant TIC98-1032-C03-01.

To obtain non-blocking behavior, it is enough for the coordinator to use a virtual synchronous uniform multicast protocol to propagate the outcome of the transaction [BT93]. This guarantees that either all or none of participants know about the fate of the transaction. Uniformity [HT93] ensures the property holds for any participant, even if it crashes during the multicast. Unfortunately, uniformity is very expensive in terms of the delay it introduces. In addition, since the delay depends on the size of the group, using uniformity seriously compromises the scalability of the protocol. To solve this two limitations, we use two different strategies. First, to increase the scalability, uniform multicast is used only within a small group of processes (the *commit servers*) instead of using it among all participants in the protocol. The idea is to employ a hierarchical configuration where a small set of processes run the protocol on behalf of a larger set of participants. Second, to minimize the latency caused by uniformity, we resort to a novel technique based on optimistic delivery that overlaps the processing of the transactional commit with the uniform delivery of the multicast. The idea here is to hide the latency of multicast behind operations that need to be performed anyway. This is accomplished by processing messages in an optimistic manner and hoping that most decisions will be correct although in some cases transactions might need to be aborted. This approach builds upon recent work in optimistic multicast [PS98] and a more aggressive version of optimistic delivery proposed in the context of Postgres-R [KPAS99] and later used to provide high performance eager replication in clusters [PJKAO0]. We use an optimistic uniform multicast that delivers messages in two steps. In the first step messages are delivered optimistically as soon as they are received. In the second step messages are delivered uniformly when they become stable. This optimistic uniform multicast is equivalent to a uniform multicast with safe indications [VKCD99].

Forced writes to the log are another source of inefficiencies in AC protocols. To guarantee correctness in case of failures, participants must flush to disk a log entry before sending their vote. This log entry contains all the information needed by a participant to recall its own actions in the event of a crash. The coordinator is also required to flush the outcome of the protocol before communicating the decision to the participants (this log entry can be skipped by using the so called *presume commit* or *presume abort* protocols [MLO86]). Flushing log records adds to the overall latency as messages cannot be sent or responded to before writing to the log. In the protocol we propose, this delay is reduced by allowing sites to send messages instead of flushing log records. The idea is to use the main memory of a replicated group (the *commit servers* mentioned above) as stable memory instead of using a mirrored log with careful writes.

Finally, to minimize the waiting time of transactions, in our protocol locks are released optimistically. The idea is that a transaction can be optimistically committed pending the confirmation provided by the uniform multicast. By optimistically committing the transaction, other transactions can proceed although they risk a rollback if the transaction that was optimistically committed ended up aborting. In our protocol, the optimistic commit is performed in such a way that aborts are confined to a single level. In addition, transactions are only optimistically committed when all their participants have voted affirmatively,



thereby greatly reducing the risk of having to abort the transaction. This contrasts with other optimistic commit protocols, e.g., [GHR97], where transactions that must abort (because one or more participants voted abort) can be optimistically committed although they will rollback anyway producing unnecessary cascading aborts.

With these properties the protocol we propose satisfactorily addresses all design concerns related to non-blocking AC and can thus become an important contribution to future distributed applications. The paper is organized as follows, Section 2 describes the system model. Section 3 and 4 present the commit algorithm and its correctness. Section 5 concludes the paper.

## 2 Model

### 2.1 Communication Model

The system consists of a set of fail-crash processes connected through reliable channels. Communication is asynchronous and by exchanging messages. A failed process can later recover with its permanent storage intact and re-join the system. Failures are detected using a (possibly unreliable) failure detector [CT96].

A virtual synchronous multicast service [BSS91,Bir96,SR93] is used. This service delivers multicast messages and views. Views indicate which processes are perceived as up and connected. We assume a virtual synchrony with the following properties: (1) *Strong virtual synchrony* [FvR95] or sending view delivery [VKCD99] that ensures that messages are delivered in the same view they were sent; (2) *Majority or primary component views* that ensure that only members within a majority view can progress, while the rest of the members block until joining again the majority view; (3) *Liveness*, when a member fails or it is partitioned from the majority view, a view excluding the failed member will be eventually delivered.

The protocol uses two different multicast primitives [HT93,SS93]: reliable (*rel-multicast*) and uniform multicasts (*uni-multicast*). Three primitives define optimistic uniform reliable multicast<sup>1</sup>: *Uni-multicast*( $m, g$ ) multicasts message  $m$  to a group  $g$ . *Opt-deliver*( $m$ ) delivers  $m$  reliably to the application. *Uni-deliver*( $m$ ) delivers  $m$  to the application uniformly. We say that a process is  $v_i$ -correct in a given view  $v_i$  if it does not fail in  $v_i$  and if  $v_{i+1}$  exists, it transits to it. The rel- and uni-multicasts preserve the following properties, where  $m$  is a message,  $g$  a group of processes and  $v_i$  a view within this group:

**OM-Validity:** If a correct process rel or uni-multicast  $m$  to  $g$  in  $v_i$ ,  $m$  will be eventually opt-delivered by every  $v_i$ -correct process.

**OM-Agreement:** If a  $v_i$ -correct process opt-delivers  $m$  in  $v_i$ , every  $v_i$ -correct process will eventually opt-deliver  $m$ .

<sup>1</sup> The failure detector must allow the implementation of the virtual synchrony model described below (e.g., the one proposed in [SR93]) and the non-blocking atomic commitment (e.g., the one in [GLS95]).

<sup>2</sup> Senders are not required to belong to the target group.

**OM-Integrity:** Any message is opt and uni-delivered by a process at most once. A message is opt-delivered only if it has been previously multicast.

Uni-multicast additionally fulfills the following properties:

**OM-Uniform-Agreement:** If a majority of  $v_i$ -correct processes opt-deliver  $m$ , they will eventually uni-deliver  $m$ . If a process uni-delivers  $m$  in  $v_i$ , every  $v_i$ -correct process will eventually uni-deliver  $m$ .

**OM-Uniform-Integrity:** A message is uni-delivered in  $v_i$  only if it was previously opt-delivered by a majority of processes in  $v_i$ .

## 2.2 Transaction Model

Clients interact with the database by issuing transactions. A transaction is a partially ordered set of *read* and *write* operations followed by either a *commit* or an *abort* operation. The decision whether to commit or abort is made after executing an optimistic atomic commitment protocol. The protocol can decide (1) to immediately abort the transaction, (2) to perform an *optimistic commit*, or (3) decide to commit or abort.

We assume a distributed database with  $n$  sites. Each site  $i$  has a *transaction manager* process  $TM_i$ . When a client submits a transaction to the system, it chooses a site as its *local* site. The local  $TM_i$  decides which other sites should get involved in processing the transaction and initiates the commitment protocol.

We assume the database uses standard mechanisms like *strict 2 phase locking* (2PL) to enforce *serializability* [BHG87]. The only change over known protocols is introduced during the commit phase of a transaction. When a transaction  $t$  is optimistically committed, all its write locks are changed to *opt* locks and all its read locks are released<sup>3</sup>. *Opt* locks are compatible with all other types of locks. That is, other transactions are allowed to set compatible locks on data held under an *opt* lock. Such transactions are said to be *on-hold*, while the rest of transactions are said to be on *normal* status. When the outcome of  $t$  is finally determined, its *opt* locks are released and all transactions that were on-hold due to these *opt* locks are returned to their normal state. A transaction that is on-hold cannot enter the commit phase until it returns to the normal state.

## 2.3 System Configuration

For the purposes of this paper, we will assume there are two disjoint groups of processes in the system. The first will conform the distributed database and will be referred as the *transaction managers* or TM group ( $TM = \{TM_1, \dots, TM_n\}$ ). Sites in this group are responsible for executing transactions and for triggering the atomic commitment protocol. By participants in the protocol, we mean processes in this group. The second group, *commit server* or CS group ( $CS = \{C_1, \dots, C_n\}$ ) is a set of replicated processes devoted to perform the AC

<sup>3</sup> Once a transaction concludes, all its read locks can be released without compromising correctness independently of whether the transaction commits or aborts [BHG87].

protocol. We assume that in any two consecutive views, there is a process that transits from the old view to the new one<sup>4</sup>.

## 2.4 Problem Definition

A non-blocking AC protocol should satisfy: (1) NBAC-Uniform validity, a transaction is (opt) committed only if all the participants voted yes; (2) NBAC-Uniform-Agreement, no two participants decide differently; (3) NBAC-Termination, if there is a time after which there is a majority view sequence in the CS group that permanently contains at least a correct process, then the protocol terminates; (4) NBAC-Non-Triviality [Gue95], if all participants voted yes, and there no failures or false suspicions, then commit is decided.

# 3 A Low Latency Commit Algorithm

## 3.1 Protocol Overview

The AC protocol starts when a client requests to commit a transaction. The commit request arrives at a transaction manager,  $TM_i$ , which then starts the protocol. The protocol involves several rounds of messages in two phases:

### First phase

1. Upon delivering the commit request,  $TM_i$  multicasts a reliable *prepare to commit* message to the TM group. This message contains the transaction identifier (*tid*) to be committed and the number of participants involved (the number of TMs contacted during the execution of the transaction).
2. Upon delivering the *prepare to commit* message, each participant unimulticasts its *vote* and the number of participants to the CS group. If a participant has not yet written the corresponding entries to its local log when the *prepare to commit* message arrives, it sends the log entry in addition to its vote without waiting to write to the log. After the message has been sent, it then writes the log entry to its local disk.

### Second phase

1. Upon *opt-delivering* a *vote* message, the processes of the commit server decide who will act as *proxy coordinator* for the protocol based on the *tid* of the transaction and the current view. Assume this site is  $C_i$ . The rest of the processes in the CS group act as backup in case  $C_i$  fails. If a no vote is opt-delivered, the transaction is aborted immediately and an *abort* message is reliable multicast to the TM group. If all votes are yes, as soon as the last vote is opt-delivered at  $C_i$ ,  $C_i$  sends a reliable multicast with an *opt-commit* message to the TM group.

---

<sup>4</sup> It might seem a strong assumption for safety that at least one server must survive between views. However, this assumption is no stronger than the usual one that assumes that the log is never lost. The strength of any of the assumptions depends on the probability of the corresponding catastrophic failures.

2. Upon delivering an *abort* message, a participant aborts the transaction. Upon delivering an *opt-commit* message, the participant changes the transaction locks to *opt* mode.
3. If all votes are affirmative, when they have been *uni-delivered* at  $C_i$ ,  $C_i$  reliable multicasts to the TM group a *commit* message.
4. When a participant delivers a *commit* or *abort* message, it releases all locks (both *opt* and non-*opt*) held by the transaction and return the corresponding transactions that were on hold to their normal state.
5. If all the votes are affirmative, the coordinator opt-commits the transaction before being excluded from the majority view (before being able to commit the transaction), and one or more votes do not reach the majority view, the transaction will be aborted by the new coordinator.

This protocol reduces the latency of the non-blocking commit in several ways. First, at no point in time in the protocol must a site wait to write a log entry to the disk before reacting to a message. The CS group acts as stable storage for both the participants (sites at the TM which could not yet write their vote and other transaction information to disk when the *prepare to commit* vote arrives) and the CS group itself (the coordinator does not need to write an entry to the log before sending the *opt-commit* message). Second, the coordinator in the CS group provides an outcome without waiting for the *vote* messages to be uniform. This reduces the overhead of uniform multicast as it overlaps its cost with that of committing the transaction.

### 3.2 The Protocol

The protocol uses the CS group to run the atomic commitment. The processes in the TM group<sup>5</sup> only act as participants and the CS group acts as coordinator. We use two tables, *trans\_tab* and *vote\_tab*, to store information in main memory about the state of a transaction and the decision of each participant regarding a given transaction at each  $CS_i$ . We also use a number of functions to change and access the values of the attributes in these tables. *Trans\_tab* contains the attributes *tid* (the transaction's identifier), *n\_participants* (number of participants in that transaction, all sites in the TM group), *timestamp* (of the first vote for timeout purposes), *coordinator* (id of the coordinator site in the CS group; this attribute is initially set with the function *store\_trans* and updated with the function *store\_coordinator*), and *outcome* (the state of the transaction; initially it is undecided, the state can be changed to aborted, opt-committed or committed by invoking the function *store\_outcome*). *Vote\_tab* contains the attributes *tid* (the transaction's identifier), *participant\_id* (site emitting the vote, which must be a site in the TM group), *vote* (the actual vote), *vote\_status* (initially optimistic, when set with the function *store\_opt\_vote*, and later definitive, when set with the function *store\_def\_vote*), and *log* (any log entry the participant may have sent with the vote). There are additional functions to consult the attributes

<sup>5</sup> For simplicity, messages are multicast to all TM processes. Processes for which the message is not relevant just discard it.

associated to each tid in the *trans\_tab*. These functions are denoted with the same name as the attribute but starting with capital letter (e.g., Timestamp). There are also functions to consult the *vote\_tab*: Log (to obtain the log sent by a participant), *N\_opt\_yes\_votes* (number of yes votes delivered optimistically for a particular transaction), *N\_def\_yes\_votes* (similarly for uni-delivered yes votes). An additional function, *Coordinator*, is used to obtain the id of the coordinator of a transaction given its tid and the current view.

### TM Group actions:

**TM.A** Upon **delivering Prepare**(tid):  
 if prepared in advance then  
   uni-multicast(CS, Vote(tid, n\_participants, my\_id, vote, empty))  
 else  
   uni-multicast(CS, Vote(tid, n\_participants, my\_id, vote, log\_record))  
 end if

**TM.B** Upon **delivering Opt-commit**(tid):  
 Change transaction tid locks to opt-mode

**TM.C** Upon **delivering Commit/Abort**(tid):  
 Commit/Abort the transaction and release transaction *tid* locks  
 Change the corresponding on-hold transactions to normal status

### CS Group actions:

**CS.A** Upon **opt-delivering Vote**(tid, n\_participants, participant\_id, vote, log):  
 store\_opt\_vote(vote\_tab, tid, participant\_id, vote, log)  
 – *the transaction outcome is still undecided*  
 if Outcome(trans\_tab, tid) = undecided then  
   if vote = no then  
     if Coordinator(current\_view, tid) = my\_id then  
       rel\_multicast(TM, Abort(tid))  
     end if  
     store\_outcome(trans\_tab, tid, aborted)  
   else – *vote = yes*  
     if N\_opt\_yes\_votes(vote\_tab, tid) = 1 then – *it is the first vote*  
       timestamp = current\_time  
       if Coordinator(current\_view, tid) = my\_id then  
         set\_up\_timer(tid, timestamp+waiting\_time)  
       end if  
       store\_trans(trans\_tab, tid, n\_participants, timestamp)  
     end if  
     if N\_opt\_yes\_votes(vote\_tab, tid) = n\_participants(trans\_tab, tid) then – *all voted yes*  
       store\_outcome(trans\_tab, tid, opt-committed)  
       if Coordinator(current\_view, tid) = my\_id then  
         disable\_timer(tid)  
         rel\_multicast(TM, Opt-commit(tid))  
       end if  
     end if  
 end if

**CS.B** Upon **uni-delivering Vote**(tid, n\_participants, participant\_id, vote, log):  
 store\_def\_vote(trans\_tab, tid, participant\_id)  
 if (N\_def\_yes\_votes(vote\_tab, tid) = n\_participants(trans\_tab, tid))  
   and (Outcome(trans\_tab, tid) ≠ abort) then  
   store\_outcome(trans\_tab, tid, committed)  
   if Coordinator(current\_view, tid) = my\_id then  
     rel\_multicast(TM, Commit(tid))  
   end if  
 end if

**CS.C** Upon **expiring Timer**(tid):  
 store\_outcome(trans\_tab, tid, aborted)  
 uni\_multicast(CS, Timeout(tid))

```

CS.D Upon uni-delivering Timeout(tid):
  store_outcome(trans_tab, tid, aborted)
  if Coordinator(current_view, tid) = my_id then
    rel_multicast(TM, Abort(tid))
  end if

CS.E Upon delivering ViewChange( $v_i$ ):
  current_view =  $v_i$ 
  – State synchronization with new members
  if my_id is the lowest in  $v_i$  that belonged to  $v_{i-1}$  then
    for every  $C_i \in v_i$  do
      if  $C_i \notin v_{i-1}$  then
        send( $C_i$ , State(trans_tab, vote_tab))
      end if
    end for
  elseif my_id  $\notin v_{i-1}$  then I am a new member
    receive(State(trans_tab, vote_tab))
  end if
  – Assignment of new coordinators in  $v_i$ 
  for each tid  $\in$  trans_tab do
    if Coordinator( $v_i$ , tid) = my_id then
      if Outcome(trans_tab, tid) = committed then
        rel_multicast(TM, Commit(tid))
      elseif Outcome(trans_tab, tid) = aborted then
        rel_multicast(TM, Abort(tid))
      else
        set_up_timer(Timestamp(trans_tab, tid) + waiting_time)
      end if
    end if
  end for
end if

```

**Dealing with coordinator failures.** Since sites in the TM group only act as participants, failures in the TM group do not affect the protocol. In the CS group, all processes are replicas of each other. Strong virtual synchrony ensures that any pending message sent in the previous view is delivered before delivering a new view. Thus, when a process fails (or it is falsely suspected), a new view is eventually delivered to a majority of available connected CS processes. Once the new view is available, a working CS process takes over as coordinator for all the on-going commitment protocols coordinated by the failed process (CS.E). For each on-going transaction commit, the new coordinator checks the delivery time of the first vote and sets up a timer accordingly (CS.E). The actions taken by the new coordinator at this point in time depend on the protocol stage. If the transaction outcome is already known (all the votes have been opt-delivered at all CS members or a no vote message has been opt-delivered), the new coordinator multicasts the outcome to the participants (CS.E). If the outcome is undecided (i.e., all previously delivered votes were affirmative and there are pending votes), the protocol proceeds as normal and the new coordinator waits until all vote messages (or a no vote) have been *opt-delivered* (CS.A).

The problematic case is when the coordinator has decided to commit the transaction and then it is excluded from the view. It can be the case that the coordinator has had time to opt-commit the transaction, but not to commit it, and that there are missing votes in the majority view. In traditional 2PC, this situation is avoided by blocking. In our protocol, the blocking situation is avoided by the use of uniform multicast within the server group. The surviving sites can safely ignore the previous coordinator: due to uniformity, at worst, they will be

aborting an opt-committed transaction, which does not violate consistency. A more accurate characterization of the rollback situation follows:

- All the votes are yes and have been opt-delivered at the coordinator.
- The coordinator successfully multicasts the opt-commit to the participants.
- The vote from at least one of the participants (e.g.,  $p$ ) is not uni-delivered.
- The coordinator and  $p$  become inoperative (e.g., due to a crash, a false suspicion, etc.) in such a way that the multicast does not reach any other member of the new primary view.

Although such a situation is possible, it is fair to say that it is extremely unlikely. Even during instability periods where false suspicions are frequent and many views are delivered, the odds for a message being opt-delivered only at the coordinator and both the coordinator and  $p$  become inoperative before the multicast of missing votes reaches any other CS member are very low. Additionally, this has to happen after the opt-commit is effective, as otherwise there would not be any rollback. Being such a rare situation, the amount of one-level aborts will be minimal even if the protocol is making optimistic decisions. It is also possible to enhance the protocol by switching optimism off during instability periods.

**Bounding commit duration.** To guarantee the liveness of the protocol and to prevent unbounded resource contention it is necessary to limit the duration of the commit phase of a transaction. This limitation is enforced by setting a timer at the coordinator when it receives the first vote from a transaction (CS.A). The rest of the members timestamp the transaction with the current time when they opt-deliver the first vote. If all participant votes have reached the coordinator before the timer expires, the timer is disabled (CS.A). Otherwise, the coordinator decides to abort the transaction but it does not immediately multicast the abort decision to the TM group (CS.C). Instead, it uni-multicasts a *timeout* message to the CS group. When this message is uni-delivered at the coordinator, a message is sent to the participants with the abort decision (CS.D).

It could be the case that the coordinator multicast a timeout message and, before uni-delivering it, the missing votes are opt-delivered at the coordinator. In that case the transaction will be aborted (its outcome is not undecided when the vote is opt-delivered since in CS.C the outcome is set to abort when the timer expires). The rest of the CS members will also abort the transaction, no matter the order in which those messages are delivered. If the missing votes are delivered before the timeout message, the transaction outcome will be set to commit (CS.A) until the timeout message is uni-delivered (if so). Upon uni-delivery of the timeout message the outcome is changed to abort (CS.D). If the timeout message is uni-delivered before the last vote at a CS member, the transaction outcome will be initially set to abort (CS.D) and remain so (CS.A).

The coordinator can be excluded from the majority view during this process and a new coordinator will take over. If the new coordinator has uni-delivered the timeout message, the outcome of the transaction will be abort (CS.E). This will happen independently of whether the old coordinator sent the abort message to the TM group. If the new coordinator has not uni-delivered the timeout

message before installing the new view, the failed coordinator did not uni-deliver it either (due to uniformity) and the new coordinator will not deliver that message (strong virtual synchrony). Therefore, the new coordinator will behave as a regular coordinator and will set the timer and wait for any pending vote (CS.E to set the timer, and CS.A in the event a vote arrives).

Despite the majority view approach, the protocol would not terminate if all the coordinators assigned to a transaction are excluded from the view before deciding the outcome. For instance, the CS group can transit perpetually between views  $\{1,2,3,4\}$  and  $\{1,2,3,5\}$ , with processes 4 and 5 being the coordinators of a transaction  $t$  in each view. In this case,  $t$  will never commit. This scenario can be avoided by, whenever possible, choosing a coordinator that has not previously coordinated the transaction. If there is at least a correct process, this will guarantee that the outcome of  $t$  will be eventually decided, thereby, ensuring the liveness of the algorithm.

**Maintaining consistency across partitions.** Although partitions always lead to blocking, our protocol maintains consistency even when partitions occur. That is, no replica decides differently on the outcome of a transaction even when the network partitions. Consistency is enforced by combining uniformity, strong virtual synchrony, and majority views. To see why, we will only consider partitions in the CS group. Partitions in the TM may lead to delays in the vote delivery (which may result in a transaction abort) and to delays in the propagation of the transaction outcome (thus, resulting in blocking during the partition). Partitions that leave the coordinator of a transaction in the majority partition of the CS group are not a problem, as the minority partition gets blocked (due to the majority view virtual synchrony). Since the transaction outcome is always decided after the uni-delivery of a message (either a vote or timeout message), uniformity guarantees that the decision will be taken by every process in the majority view. When the coordinator of a transaction is in a minority partition, undecided transactions cannot create problems as the coordinator, once in the minority partition, will block. When this happens, a new coordinator can make any decision regarding undecided transactions without compromising consistency. Only transactions whose outcome has been decided by the coordinator during the partition may lead to inconsistencies. There are four cases to consider:

- The coordinator optimistically commits a transaction when it opt-delivers the last vote (and all votes have been affirmative). Assume a partition leaves the coordinator in a minority partition. The new coordinator may (1) opt-deliver all the votes or (2) never deliver one or more votes. In the first case, it will opt-commit the transaction (CS.A) thereby agreeing with the old coordinator. In the second case, it will abort the transaction once the timer expires (CS.C). Since the transaction was only optimistically committed by the old coordinator, the new coordinator is free to decide to abort without violating consistency.
- If the old coordinator committed a transaction, the new coordinator will do the same. A transaction is committed when all the votes have been uni-



delivered (CS.B). If the votes were uni-delivered at the old coordinator, all the processes in that view also uni-delivered them in that view (uniformity and strong virtual synchrony). Thus, the new coordinator will also commit the transaction (CS.E).

- The old coordinator opt-delivered a no vote and aborted the transaction. The new coordinator will either have delivered the no vote or timed out. In both cases the new coordinator will also abort the transaction (CS.A and CS.C, respectively) thereby agreeing with the old coordinator.
- The old coordinator timed out and aborted the transaction. The transaction will not be effectively aborted until the timeout message is uni-delivered. Uniformity guarantees that the timeout message, if uni-delivered, will be uni-delivered to both the old and the new coordinator, thereby preventing any inconsistency.

**Replica recovery and partition merges.** In order, to maintain an appropriate level of availability, it is necessary to enable new (or recovered) replicas to join the CS group and to allow partitioned groups to merge again. When a new process joins the CS group, virtual synchrony guarantees that the new process will deliver all the messages delivered by the other replicas after installing the new view (and thus, after state synchronization). The installation of the new view will trigger state synchronization (CS.E). This involves sending from an old member of the group (one that transits from the previous view to the current one, that it is guaranteed to exist due to the majority view approach) a *State* message with the `vote_tab` and the `trans_tab` tables to the new member. Members from a minority partition that join a majority view will be treated as recovered members, that is, they will be sent the up-to-date tables from a process belonging to the previous majority view.

The state transfer and the assumption that at least a process from the previous view transits to the next view guarantees that a new member acting as coordinator will use up-to-date information, thereby ensuring the consistency of the protocol. The recovery of a participant has not been included in the algorithm due to its simplicity (upon recovery it will just ask to the CS group about the fate of some transactions).

## 4 Correctness

**Lemma 1 (NBAC-Uniform-Validity).** *A transaction (opt) commits only if all the participants voted yes.* □

**Proof** (lemma II): (Opt) Commit is decided when the coordinator multicasts such message after the transaction has been recorded as (opt) committed in the `trans_tab` (CS.A). This can only happen when all participant votes have been (opt) uni-delivered and they are yes votes. □

**Lemma 2 (NBAC-Uniform-Agreement).** *No two CS members decide differently.* □

**Proof** (lemma 2): In the absence of failures the lemma is proved trivially, as only the coordinator decides about the outcome of the transaction. The rest of them just logs the information about the transaction in case they have to take over. The only way for two members to decide on the same transaction is that one is a coordinator of the transaction and then it is excluded from the majority view before deciding the outcome. Then, a CS member takes over as new coordinator and decides about the transaction.

Assume that a coordinator makes a decision and due to its exclusion from view  $v_i$ , a new coordinator takes over and makes a different decision. Let us assume without loss of generality that the new coordinator takes over in view  $v_{i+1}$  (in general, it will be in  $v_{i+f}$ ). The old coordinator can have decided to:

1. *Commit*. The old coordinator can only decide commit if all votes have been uni-delivered, and they all were affirmative. If the new coordinator decides to abort, it can only be because it has not uni-delivered one or more votes neither before the view change nor before its timer expires. In this situation, there are two cases to consider:
  - a. The new coordinator belonged to  $v_i$ . Hence, all votes were uni-delivered in  $v_i$  at the old coordinator (which needed all yes votes to decide to commit) but not at the new coordinator (otherwise it would also decide to commit) what violates multicast uniformity.
  - b. The new coordinator joined the CS group after a recovery or a partition merge. From the recovery procedure, the new coordinator has gotten the most up-to-date state during the state transfer triggered by the view change. If it decides to abort, it is because a process in view  $v_i$  (the one which sent its tables in the state transfer) did not uni-deliver one or more votes. This again violates uniformity and it is therefore impossible.
2. *Abort due to a no vote*. If the old coordinator aborts the transaction, it does so as soon as the no vote is opt-delivered (CS.A). In order to decide to commit, the new coordinator needs to uni-deliver all votes and that all votes are yes. Since a participant votes only once, this situation cannot occur (for it to occur, a participant needs to say no to the old coordinator and yes to the new one).
3. *Abort due to a timeout*. If the old coordinator decided to abort due to a timeout, then it uni-delivered its own timeout message (CS.C). If the new coordinator decides to commit, then it must have uni-delivered all the votes before its timer expires and before uni-delivering the timeout message. This implies that the timeout message has been delivered to the old coordinator in view  $v_i$  and not to the new coordinator. Now there are two cases to consider:
  - a. If the new coordinator was in view  $v_i$ , the fact that the old coordinator has not received the timeout message violates multicast uniformity. It is therefore not possible for the new coordinator to have been in  $v_i$ .
  - b. If the new coordinator was not in view  $v_i$  then it has joined the group in view  $v_{i+1}$ , and thus during the state synchronization it has received the most up-to-date tables. However, this implies that some process in the CS group was in view  $v_i$ , transited to  $v_{i+1}$ , but did not uni-deliver the timeout message. Again this violates uniformity.

From here, since in all possible cases the new coordinator cannot make a different decision than the old coordinator once the latter has made a decision (abort or commit), the lemma is proven.  $\square$

**Lemma 3 (NBAC-Termination).** *If there is a time after, which there is a majority view sequence in the CS group that permanently contains at least a correct process, then the protocol terminates.*  $\square$

**Proof** (lemma 3): Assume for contradiction that the protocol never ends. This means that either:

- A correct coordinator never decides. A correct coordinator will either: (1) Opt-deliver a no vote, in which case the transaction is aborted, or (2) uni-deliver all the votes and are all yes, in which case the transaction is committed, or (3) uni-deliver the timeout message before opt-delivering all the votes (and being all previous votes affirmative), in which case the transaction is aborted. Therefore, if there is a correct process, there will eventually be a correct coordinator that will decide the transaction outcome and multicast it to the participants, thus terminating the protocol.
- There is an infinite sequence of unsuccessful coordinators that do not terminate the protocol. The NewCoordinator function, whenever possible, chooses a fresh coordinator (a process that did not previously coordinate the transaction). This means that a correct process  $p$  will eventually coordinate the transaction. Since  $p$  is correct and belongs to the majority view, it will eventually terminate the protocol as shown before.

Thereby, it is proven that the protocol eventually terminates.  $\square$

**Lemma 4 (NBAC-Non-Triviality).** *If all participants votes yes there are no failures nor false suspicions then commit will be decided.*  $\square$

**Proof** (lemma 4): The abort decision can only be taken when the coordinator receives a no vote, or because it times out. Otherwise, the decision is commit.  $\square$

**Theorem 1 (NBAC-Correctness).** *The protocol presented in the paper fulfills the non-blocking atomic commitment properties: NBAC-Validity, NBAC-Uniform-Agreement, NBAC-Termination, and NBAC-Non-Triviality.*  $\square$

**Proof** (theorem 1): It follows from lemmas 1, 2, 3, and 4.  $\square$

## 5 Conclusions

Atomic commitment is an important feature in distributed transactional systems. Many commercial products and research prototypes use it to guarantee transactional atomicity (and, with it, data consistency) across distributed applications. The current standard protocol for atomic commitment is 2PC which offers reasonable performance but might block when certain failures occur. In this paper we have proposed a non-blocking atomic commitment protocol that

offers the same reasonable performance as 2PC but that is non-blocking. Unlike previous work in the area, we have emphasized several practical aspects of atomic commitment. First, the new protocol does not create any additional message overhead when compared with 2PC. Second, by using a replicated group as stable memory instead of having to flush log records to the disk, the protocol is likely to exhibit a shorter response time than standard 2PC. Third, the fact that the second round of the commit protocol is run only by a small subset of the participants minimizes the overall overhead. Fourth, and most relevant in practice, the new protocol can be implemented on top of the same interface as that used for 2PC. This is because, unlike most non-blocking protocols that have been previously proposed, the participants only need to understand a *prepare to commit* message (or *vote-request*) and then a *commit* or *abort* message. This is exactly the same interface required for 2PC and it is implemented in all transactional applications. Because of these features, we believe the protocol constitutes an important contribution to the design of distributed transactional systems. We are currently evaluating the protocol empirically to get performance measures and are looking into several possible implementations to further demonstrate the advantages it offers.

## References

- [BHG87] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, Reading, MA, 1987.
- [Bir96] K.P. Birman. *Building Secure and Reliable Network Applications*. Prentice Hall, NJ, 1996.
- [BSS91] K. P. Birman, A. Schiper, and P. Stephenson. Lightweight Causal and Atomic Group Multicast. *ACM Transactions on Computer Systems*, 9(3):272–314, August 1991.
- [BT93] O. Babaoglu and S. Toueg. Understanding Non-Blocking Atomic Commitment. In *Distributed Systems*. Addison Wesley, 1993.
- [CT96] T. D. Chandra and S. Toueg. Unreliable Failure Detectors for Reliable Distributed Systems. *Journal of the ACM*, 43(2):225–267, March 1996.
- [DS83] C. Dwork and D. Skeen. The Inherent Cost of Nonblocking Commitment. In *Proc. of ACM PODC*, pages 1–11, 1983.
- [FvR95] R. Friedman and R. van Renesse. Strong and Weak Virtual Synchrony in Horus. Technical report, CS Dep., Cornell Univ., 1995.
- [GHR97] R. Gupta, J. Haritsa, and K. Ramamritham. Revisiting Commit Processing in Distributed Database Systems. In *Proc. of the ACM SIGMOD*, 1997.
- [GLS95] R. Guerraoui, M. Larrea, and A. Schiper. Non-Blocking Atomic Commitment with an Unreliable Failure Detector. In *Proc. of the 14th IEEE Symp. on Reliable Distributed Systems*, Bad Neuenahr, Germany, September 1995.
- [GLS96] R. Guerraoui, M. Larrea, and A. Schiper. Reducing the Cost for Non-Blocking in Atomic Commitment. In *Proc. of IEEE ICDCS*, 1996.
- [Gra78] J. Gray. *Notes on Data Base Operating Systems. Operating Systems: An Advanced Course*. Springer, 1978.

- [GS95] R. Guerraoui and A. Schiper. The Decentralized Non-Blocking Atomic Protocol. In *Proc. of IEEE SPDP*, 1995.
- [Gue95] R. Guerraoui. Revisiting the Relationship Between Non-Blocking Atomic Commitment and Consensus. In *Proc. of the WDAG'95*, 1995.
- [HT93] V. Hadzilacos and S. Toueg. Fault-Tolerant Broadcasts and Related Problems. In *Distributed Systems*, pages 97–145. Addison Wesley, 1993.
- [KD95] I. Keidar and D. Dolev. Increasing the Resilience of Atomic Commit at No Additional Cost. In *Proc. of ACM PODS*, 1995.
- [KPAS99] B. Kemme, F. Pedone, G. Alonso, and A. Schiper. Processing Transactions over Optimistic Atomic Broadcast Protocols. In *Proc. of 19th IEEE Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 424–431, 1999.
- [KR01] I. Keidar and S. Rajsbaum. On the Cost of Fault-Tolerant Consensus Where There No Faults - A Tutorial. Technical Report MIT-LCS-TR-821, 2001.
- [MLO86] C. Mohan, B. Lindsay, and R. Obermark. Transaction Management in the R\* Distributed Database System. *ACM Transactions on Database Systems*, 11(4), February 1986.
- [PJKAO0] M. Patiño Martínez, R. Jiménez Peris, B. Kemme, and G. Alonso. Scalable Replication in Database Clusters. In *Proc. of Distributed Computing Conf., DISC'00. Toledo, Spain*, volume LNCS 1914, pages 315–329, October 2000.
- [PS98] F. Pedone and A. Schiper. Optimistic Atomic Broadcast. In S. Kuten, editor, *Proc. of 12th Distributed Computing Conference*, volume LNCS 1499, pages 318–332. Springer, September 1998.
- [Ske81] D. Skeen. Nonblocking Commit Protocols. *ACM SIGMOD*, 1981.
- [Ske82] D. Skeen. A Quorum-Based Commit Protocol. In *Proc. of the Works. on Distributed Data Management and Computer Networks*, pages 69–80, 1982.
- [SR93] A. Schiper and A. Ricciardi. Virtually Synchronous Communication Based on a Weak failure Susceptor. In *Proc. of FTCS-23*, pages 534–543, 1993.
- [SS93] A. Schiper and A. Sandoz. Uniform Reliable Multicast in a Virtually Synchronous Environment. In *Proc. of ICDCS-13*, pages 561–568, 1993.
- [VKCD99] R. Vitenberg, I. Keidar, G. V. Chockler, and D. Dolev. Group Communication Specifications: A Comprehensive Study. Technical Report CS99-31, Hebrew Univ., September 1999. To be published in ACM Comp. Surveys.

# Stable Leader Election

## (Extended Abstract)

Marcos K. Aguilera<sup>1</sup>, Carole Delporte-Gallet<sup>2</sup>, Hugues Fauconnier<sup>2</sup>, and Sam Toueg<sup>3</sup>

<sup>1</sup> Compaq Systems Research Center, 130 Lytton Ave, Palo Alto, CA, 94301, USA

Marcos.Aguilera@compaq.com

<sup>2</sup> LIAFA, Université D. Diderot, 2 Place Jussieu, 75251, Paris Cedex 05, France

{cd,hf}@liafa.jussieu.fr

<sup>3</sup> DIX, École Polytechnique, 91128 Palaiseau Cedex, France

sam@lix.polytechnique.fr

**Abstract.** We introduce the notion of *stable* leader election and derive several algorithms for this problem. Roughly speaking, a leader election algorithm is stable if it ensures that once a leader is elected, it remains the leader for as long as it does not crash and its links have been behaving well, irrespective of the behavior of other processes and links. In addition to being stable, our leader election algorithms have several desirable properties. In particular, they are all communication-efficient, i.e., they eventually use only  $n$  links to carry messages, and they are robust, i.e., they work in systems where only the links to/from some correct process are required to be eventually timely. Moreover, our best leader election algorithm tolerates message losses, and it ensures that a leader is elected in constant time when the system is stable. We conclude the paper by applying the above ideas to derive a robust and efficient algorithm for the eventually perfect failure detector  $\diamond P$ .

## 1 Introduction

### 1.1 Motivation and Background

Failure detection is at the core of many fault-tolerant systems and algorithms, and the study of failure detectors has been the subject of intensive research in recent years. In particular, there is growing interest in developing failure detector implementations that are efficient, timely, and accurate [VRMMH98, LAF99, CTA00, FRT00, LFA00b].

A failure detector of particular interest is  $\Omega$  [CHT96]. At every process  $p$ , and at each time  $t$ , the output of  $\Omega$  at  $p$  is a single process, say  $q$ . We say that  $p$  *trusts*  $q$  to be up at time  $t$ .  $\Omega$  ensures that eventually all correct processes trusts the *same* process and that this process is *correct*.

Note that a failure detector  $\Omega$  can also be thought of as a leader elector: The process currently trusted to be up by  $p$ , can be thought of as the current “leader” of  $p$ , and  $\Omega$  ensures that eventually all processes have the same leader.

An  $\Omega$  leader election is useful in many settings in distributed systems. For example, some algorithms use it to solve consensus in asynchronous systems with failures [Lam98, MR00, LFA00a] (in fact,  $\Omega$  is the weakest failure detector to solve consensus [CHT96]). Electing a leader can also be useful to solve a set of tasks efficiently in distributed

environments [DHW98]. Even though  $\Omega$  is strong enough to solve hard problems such as consensus, we will see that it is weak enough to admit efficient implementations.

Our main goal here is to propose efficient algorithms for  $\Omega$  in partially synchronous systems with process crashes and message losses. To illustrate this problem, consider the following simple implementation of  $\Omega$  [Lam98, DPLL97]. Assume that processes may crash, but *all* links are eventually timely, i.e., there is a time after which all messages sent take at most  $\delta$  time to be received. In this system, one can implement  $\Omega$  as follows:

1. Every process  $p$  periodically sends an OK message to all, and maintains a set of processes from which it received an OK recently.<sup>1</sup>
2. The output of  $\Omega$  at  $p$  is simply the smallest process currently in  $p$ 's set.

Note that the set of processes that  $p$  builds in part (1) is eventually equal to the set of all correct processes. Thus part (1) actually implements an *eventually perfect failure detector*  $\diamond P$  [CT96]. So, in the above algorithm, we implement  $\Omega$  by first implementing  $\diamond P$  and then outputting the smallest process in the set of processes trusted by  $\diamond P$ .

This implementation of  $\Omega$  has several drawbacks:

1. *The system assumptions required by the algorithm are too strong.* In fact, this algorithm works only if *all* links are eventually timely. Intuitively, however, one should be able to find a leader in systems where only the links to and from some correct process are eventually timely. In other words, while this algorithm requires  $n^2$  eventually timely links, we would like an algorithm that works even if there are only  $n$  eventually timely links (those to and from some correct process).
2. *The algorithm is not communication-efficient.* In this algorithm, every process sends an OK message to all processes, forever. That is, all the  $n^2$  links carry messages, in both directions, forever. Intuitively, this is wasteful: once a correct process is elected as a leader, it should be sufficient for it to periodically send OK messages to all processes (to inform them that it is still alive and so they can keep it as their leader), and all other processes can keep quiet. In other words, after an election is over, no more than  $n$  links should carry messages (those links from the elected leader to the other processes). All the other links should become quiescent. We say that a leader election algorithm is *communication-efficient* if there is a time after which it uses only unidirectional  $n$  links.
3. *The election is not stable.* In this algorithm, processes can demote their current leader and elect a new leader for no real reason: even if the current leader has not crashed and its links have been timely for an (arbitrarily) long time, the leader can still be demoted at any moment by an extraneous event. To see this, suppose process 2 is trusted forever by  $\diamond P$  (because it is correct and all its links are timely) and that it is the current leader (because it is the smallest process currently trusted by  $\diamond P$ ). If  $\diamond P$  starts trusting process 1 (this can occur if the links from process 1 become timely), then 2 loses the leadership and 1 is elected. If later 1 is suspected again, 2 regains the leadership. So 2 loses the leadership each time 1 becomes “good”

<sup>1</sup> “Recent” means within  $\Delta$  from the last OK received. If processes send OK every  $\eta$  then  $\Delta = \delta + \eta$ . If  $\delta$  and  $\eta$  are not known,  $p$  sets  $\Delta$  by incrementing it for every mistake it makes [CT96].

again, *even though 2 keeps behaving well and remains trusted forever by all the processes!* This is a serious drawback, because leadership changes are quite costly to most applications that rely on a leader. Thus, we are seeking a *stable* leader election algorithm. Roughly speaking, such an algorithm ensures that once a leader is elected, it remains the leader for as long as it does not crash and its links have been behaving well, irrespective of the behavior of other processes or links.

Our main goal here is to give algorithms for  $\Omega$  (i.e., leader election algorithms) that are communication-efficient, stable, and that work in systems where only the links to and from some correct process are required to be eventually timely, as explained above. In addition, we want an algorithm for  $\Omega$  that can elect a leader quickly when the system “stabilizes”, i.e., it has a small *election time*.

We achieve our goal progressively. We first present an algorithm for  $\Omega$  that is communication-efficient. This algorithm is simple, however it has the following drawbacks: (a) it is not stable, (b) it assumes that messages are not lost and (c) its worst-case election time is proportional to  $n$ ,<sup>2</sup> even when the system is stable. We next modify this algorithm to achieve stability. Then we change it so that it works despite message losses. Finally, we modify it to achieve constant election time when the system “stabilizes”. It is worth noting that our algorithms are self-stabilizing.

We conclude the paper by using our techniques to give an algorithm for  $\Diamond P$  that is both robust and efficient: In contrast to previous implementations of  $\Diamond P$ , our algorithm works in systems where only  $n$  bidirectional links are required to be eventually timely, and there is a time after which only  $n$  bidirectional links carry messages. This algorithm for  $\Diamond P$  works despite message losses.

## 1.2 Related Work

The simple implementation of  $\Omega$  described above is mentioned in several works (e.g., [Lam98,DPLL97]). Such an implementation, however, requires strong systems assumptions, is not communication-efficient, and is not stable. Larrea et al. give an algorithm for  $\Omega$  that is communication-efficient, but it requires strong systems assumptions, and is not stable [LFA00b]. An indirect way to implement  $\Omega$  is to first implement an eventually strong failure detector  $\Diamond S$  [CT96] and then transform it into  $\Omega$  using the algorithms in [Chu98]. But such implementations also have drawbacks. First, the known implementations of  $\Diamond S$  are either not communication-efficient [CT96,ACT99,ACT00] or they require strong system assumptions [LAF99,LFA00b]. Second, the  $\Omega$  that we get this way is not necessarily stable.

To the best of our knowledge, all prior implementations of  $\Diamond P$  require that  $O(n^2)$  links to be eventually timely. Larrea et al propose a communication-efficient transformation of  $\Omega$  to  $\Diamond P$ , but it requires all links to be eventually timely and it does not tolerate message losses [Lar00].

## 1.3 Summary of Contributions

The contributions of the paper are the following:

<sup>2</sup> Actually, it is proportional to the maximum number of failures.



- We introduce the notion of *stable* leader election and describe the first leader election algorithm that is simultaneously stable and communication-efficient and requires only  $n$  eventually timely bidirectional links.
- We modify our algorithm to work with message losses, first when processes have approximately synchronized clocks, and then when clocks are drift-free or have bounded drift.
- We show how to achieve constant election time during good system periods.
- We give an algorithm for  $\diamond P$  that is both robust and efficient: it works in systems where only  $n$  bidirectional links are required to be eventually timely, and there is a time after which only  $n$  bidirectional links carry messages.

## 1.4 Roadmap

This paper is organized as follows. In Sect. 2 we describe our model. In Sect. 3, we define the problem of stable and message-efficient  $\Omega$  leader election. In Sect. 4, we give a simple algorithm for  $\Omega$ , and then modify it in Sect. 5 to make it stable. In Sect. 6, we give algorithms for  $\Omega$  that work despite message losses. Then, in Sect. 7, we explain how to obtain  $\Omega$  with a constant election time when the system stabilizes. In Sect. 8, we discuss view numbers. Finally, in Sect. 9, we give a new algorithm for  $\diamond P$  that guarantees that, there is a time after which, only  $n$  bidirectional links carry messages.

Because of space limitations in this extended abstract, we have omitted all technical proofs. They can be found in [ADGET01].

## 2 Informal Model

We consider a distributed system with  $n \geq 2$  processes  $\Pi = \{0, \dots, n-1\}$  that can communicate with each other by sending messages through a set of links  $\Lambda$ . We assume that the network is fully connected, i.e.,  $\Lambda = \Pi \times \Pi$ . The link from process  $p$  to process  $q$  is denoted by  $p \rightarrow q$ . The system is partially synchronous in that (1) links are sometimes timely (good) and sometimes slow, (2) processes have drift-free clocks (which may or may not be synchronized) and (3) there is an upper bound  $B$  on the time a process takes to execute a step. For simplicity, we assume that  $B = 0$ , i.e., processes execute a step instantaneously, but it is easy to modify our results for any  $B \geq 0$ . At each step a process can (1) receive a message, (2) change its state and (3) send a message. The value of a variable of a process at time  $t$  is the value of that variable after the process takes a step at time  $t$ .

**Processes and process failure patterns.** Processes can fail by crashing, and crashes are permanent. A process failure pattern is function  $F_P$  that indicates, for each time  $t$ , what processes have crashed by  $t$ . We say that *process  $p$  is alive at time  $t$*  if  $p \notin F_P(t)$ . We say process  $p$  is correct if it is always alive.

**Link behavior pattern.** A link behavior pattern is a function  $F_L$  that determines, for each time  $t$ , which links are good at  $t$ . The guarantees provided by links when they are good are specified by axiomatic links properties. These properties, given below, depend on whether the link is reliable or lossy.

**Reliable links.** Some of our basic algorithms require reliable links. Such links do not create, duplicate or drop messages. The link may sometimes be good and sometimes slow. If a process sends a message through a link and the link remains good for  $\delta$  ( $\delta$  is a system parameter known by processes) then the recipient gets the message within  $\delta$ . More precisely, a reliable link  $p \rightarrow q \in \Lambda$  satisfies the following properties:

- (No creation or duplication): Process  $q$  receives a message  $m$  from  $p$  at most once, and only if  $p$  previously sent  $m$  to  $q$ .
- (No late delivery in good periods): If  $p$  sends  $m$  to  $q$  by time  $t - \delta$  and  $p \rightarrow q$  is good during times  $[t - \delta, t]$  then  $q$  does not receive  $m$  after time  $t$ .
- (No loss): If  $p$  sends  $m$  to  $q$  then  $q$  eventually receives  $m$  from  $p$ .<sup>3</sup>

**Lossy links.** Like reliable links, lossy links do not create or duplicate messages and may be slow or not. However, unlike reliable links, they may drop messages when they are not good. A lossy link  $p \rightarrow q \in \Lambda$  satisfies the following properties:

- (No creation or duplication): Same as above.
- (No late delivery in good periods): Same as above.
- (No loss in good periods): If  $p$  sends  $m$  to  $q$  at time  $t - \delta$  and  $p \rightarrow q$  is good during times  $[t - \delta, t]$  then  $q$  eventually receives  $m$  from  $p$ .

**Connectivity.** In this paper, we focus on implementing  $\Omega$  (defined below). It is easy to show that this is impossible if links are never good. We thus assume that there exists at least one process whose links are eventually good. More precisely, we say that a process  $p$  is *accessible at time  $t$*  if it is alive at time  $t$  and all links to and from  $p$  are good at time  $t$ .<sup>4</sup> We say that  $p$  is *eventually accessible* if there exists a time  $t$  such that  $p$  is accessible at every time after  $t$ .<sup>5</sup> We assume that there exists at least *one* process that is eventually accessible.

### 3 Stable Leader Election

#### 3.1 Specification of $\Omega$

We consider a weak form of leader election, denoted  $\Omega$ , in which each process  $p$  has a variable  $leader_p$  that holds the identity of a process or  $\perp$ .<sup>6</sup> Intuitively, eventually all alive processes should hold the identity of the same process, and that process should be correct. More precisely, we require the following property:

- There exists a correct process  $\ell$  and a time after which, for every alive process  $p$ ,  $leader_p = \ell$ .

<sup>3</sup> For convenience, in our model dead processes “receive” messages that are sent to them (but of course they cannot process such messages).

<sup>4</sup> For convenience, we assume that a process is not accessible at times  $t < 0$ .

<sup>5</sup> Note that eventually-forever accessible would be a more precise name for this property, but it is rather cumbersome.

<sup>6</sup> The original definition of  $\Omega$  does not allow the output to be  $\perp$ . We allow it here because it is convenient for processes to know when the leader elector has not yet selected a leader.

If at time  $t$ ,  $leader_p$  contains the same process  $\ell$  for all alive processes  $p$ , then we say that  $\ell$  is leader at time  $t$ . Note that a process  $p$  never knows if  $leader_p$  is really the leader at time  $t$ , or not. A process only knows that *eventually*  $leader_p$  is leader. This guarantee seems rather weak, but it is actually quite powerful: it can be used to solve consensus in asynchronous systems [CHT96].

### 3.2 Communication-Efficiency

An algorithm for  $\Omega$  is communication-efficient if there is a time after which it uses only  $n$  unidirectional links. All our  $\Omega$  algorithms are communication-efficient. Actually, if we discount messages from a process to itself, our algorithms use only  $n - 1$  links, which is optimal [LFA00b].

### 3.3 Stability

A change of leadership often incurs overhead to an application, which must react to deal with the new leader. Thus, we would like to avoid switching leaders as much as possible, unless there is a good reason to do so. For instance, if the leader has died or has been inaccessible to processes, it must be replaced. An algorithm that changes leader *only* in those circumstances is called *stable*. More precisely, a  $k$ -stable algorithm guarantees that in every run,

- if  $p$  is leader at time  $t$  and  $p$  is accessible during times  $[t - k\delta, t + 1]$  then  $p$  is leader at time  $t + 1$ .

Here,  $k$  is a parameter that depends on the algorithm; the smaller the  $k$ , the better the algorithm because it provides a stronger stability property. We introduced parameter  $k$  because no algorithm can be “instantaneously” stable (0-stable) and 1-stable algorithms have serious drawbacks, as we show in the full paper [ADGFT01]. Our algorithm for reliable links is 3-stable while our best algorithm for lossy links is 6-stable.

## 4 Basic Algorithm for $\Omega$

Figure 1 shows an algorithm for  $\Omega$  that works in systems with reliable links. This algorithm is simple and communication-efficient but not stable — we will later modify it to get stability. Intuitively, processes execute in rounds  $r = 0, 1, 2, \dots$ , where variable  $r$  keeps the process’s current round. To start a round  $k$ , a process (1) sends  $(START, k)$  to a specially designated process, called the “leader of round  $k$ ”; this is just process  $k \bmod n$ , (2) sets  $r$  to  $k$ , (3) sets the output of  $\Omega$  to  $k \bmod n$  and (4) starts a timer — a variable that is automatically incremented at each clock tick. While in round  $r$ , the process checks if it is the leader of that round (task 0) and if so sends  $(OK, r)$  to all every  $\delta$  time. When a process receives an  $(OK, k)$  for the current round ( $r = k$ ), the process

<sup>7</sup> In this and other algorithms, we chose the sending period to be equal to the network delay  $\delta$ . This arbitrary choice was made for simplicity of presentation only. In general, the sending period can be set to any value  $\eta$ , though, in this case, one needs to modify the algorithms slightly, e.g., by adjusting the time out periods. The choice of  $\eta$  affects the quality of service of the failure detector [CTA00], such as how fast a leader is demoted if it crashes.

---

Code for each process  $p$ :

```

1  procedure  $StartRound(s)$                                 { executed upon start of a new round }
2    if  $p \neq s \bmod n$  then send  $(START, s)$  to  $s \bmod n$       { wake up new leader }
3     $r \leftarrow s$                                            { update current round }
4     $leader \leftarrow s \bmod n$                                { output of  $\Omega$  }
5    restart timer

6  on initialization:
7     $StartRound(0)$ 
8    start tasks 0 and 1

9  task 0:                                                    { leader sends  $OK$  every  $\delta$  time }
10   loop forever
11     if  $p = r \bmod n$  and have not sent  $(OK, r)$  within  $\delta$  then send  $(OK, r)$  to all

12  task 1:
13   upon receive  $(OK, k)$  with  $k = r$  do                      { current leader is active }
14     restart timer

15   upon timer  $> 2\delta$  do                                    { timeout on current leader }
16      $StartRound(r + 1)$                                        { start next round }

17   upon receive  $(OK, k)$  or  $(START, k)$  with  $k > r$  do
18      $StartRound(k)$                                            { jump to round  $k$  }

```

---

**Fig. 1.** Basic algorithm for  $\Omega$  with reliable links.

---

restarts its timer. If the process does not receive  $(OK, r)$  for more than  $2\delta$  time, it times out on round  $r$  and starts round  $r + 1$ . If a process receives  $(OK, k)$  or  $(START, k)$  from a higher round ( $k > r$ ), the process starts that round.

Intuitively, this algorithm works because it guarantees that (1) if the leader of the current round crashes then the process starts a new round and (2) processes eventually reach a round whose leader is a correct process that sends timely  $(OK, k)$  messages.

**Theorem 1.** *Consider a system with reliable links. Assume some process is eventually accessible. Figure 1 is a communication-efficient algorithm for  $\Omega$ .*

## 5 Stable Algorithm for $\Omega$

The algorithm of Fig. 1 implements  $\Omega$  but it is not stable because it is possible that (1) some process  $q$  is accessible for an arbitrarily long time, (2) all alive processes have  $q$  as their leader at time  $t$ , but (3)  $q$  is demoted at time  $t + 1$ . This could happen in two essentially different ways:

**Problem scenario 1.** Initially all processes are in round 0 and so process 0 is the leader. All links are good (timely), except the links to and from process 2, which are very slow. Then at time  $2\delta + 1$ , process 2 times out on round 0 and starts round 1, and so 0 loses leadership. Moreover, 2 sends  $(START, 1)$  to process 1. At time  $2\delta + 2$ , process 2

crashes and process 0 becomes accessible. Message  $(START, 1)$  is delayed until some arbitrarily large time  $M \gg 2\delta + 2$ . At time  $M$ , process 1 receives  $(START, 1)$  and starts round 1, and thus process 0 is no longer leader.

In this scenario, process 0 becomes the leader right after process 2 crashes, since all alive processes are then in round 0 and hence have 0 as their leader. Unfortunately 0 is demoted at time  $M$ , even though it has been accessible to all processes during the arbitrarily long period  $[2\delta + 2, M]$ .

**Problem scenario 2.** We divide this scenario in two stages. (A) *Setup stage*. Initially process 1 times out on round 0, starts round 1 and sends  $(OK, 1)$  to all. All processes except process 0 get this message and start round 1. Then process 3 times out on rounds 1 and 2, starts round 3 and sends  $(OK, 3)$  to all. All processes except process 0 get that message and start round 3; process 0, however, remains in round 0 (because all messages from higher rounds are delayed). Then process 2 becomes accessible. All processes except 0 remain in round 3 for a long time, while 0 remains in round 0 for a long time. All processes except 0 then progressively time out on rounds 3, 4,  $\dots$  until they start round  $n + 2$ , say at time  $t$ . Meanwhile, process 0 receives the old  $(OK, 1)$  message, advances to round 1, timeouts on round 1 and starts round 2 at time  $t$ . (B) *Demote stage*. Note that at time  $t$ , process 2 is the leader because all processes are in a round congruent to 2 modulo  $n$ . Moreover, 2 has been accessible for a long time. Unfortunately, process 2 stops being the leader when process 0 receives  $(OK, 3)$  and starts round 3.

**Summary of bad scenarios.** Essentially, scenario 1 is problematic because a single process may (1) time out on the current round, (2) send a message to move to a higher round and then (3) die. This message may be delayed and may demote the leader long in the future. On the other hand, scenario 2 is problematic because a process may become a leader while processes are in different rounds; after the leader is elected, a process in a lower round may switch its leader by moving to a higher round.

Our new algorithm, shown in Fig. 2, avoids the above problems. To prevent problem scenario 1, when a process times out on round  $k$ , it sends a  $(STOP, k)$  message to  $k \bmod n$  before starting the next round. When  $k \bmod n$  receives such a message, it abandons round  $k$  and starts round  $k + 1$ . To see why this prevents scenario 1, note that, before process 2 sends  $(START, 1)$  to 1, it sends  $(STOP, 0)$  to 0. Soon after process 0 becomes accessible, it receives such a message and abandons round 0.

To avoid problem scenario 2, when a process starts round  $k$ , it no longer sets *leader* to  $k \bmod n$ . Instead, it sets it to  $\perp$  and waits until it receives two  $(OK, k)$  messages from  $k \bmod n$ . Only then it sets *leader* to  $k \bmod n$ . This guarantees that if  $k \bmod n$  is accessible and some process sets *leader* to  $k \bmod n$ , then all processes have received at least one  $(OK, k)$  and hence have started round  $k$ . In this way, all processes are in the same round  $k$ .

**Theorem 2.** Consider a system with reliable links. Assume some process is eventually accessible. Figure 2 is a 3-stable communication-efficient algorithm for  $\Omega$ .

---

Code for each process  $p$ :

```

1  procedure StartRound( $s$ )                                { executed upon start of a new round }
2    if  $p \neq s \bmod n$ 
3    then send (START,  $s$ ) to  $s \bmod n$                     { wake up the new leader candidate }
4     $r \leftarrow s$                                           { update current round }
5     $leader \leftarrow \perp$                                 { demote previous leader but do not elect leader quite yet }
6    restart timer

7  on initialization:
8    StartRound(0)
9    start tasks 0 and 1

10 task 0:                                                  { leader/candidate sends OK every  $\delta$  time }
11   loop forever
12   if  $p = r \bmod n$  and have not sent (OK,  $r$ ) within  $\delta$  then send (OK,  $r$ ) to all

13 task 1:
14   upon receive (OK,  $k$ ) with  $k = r$  do                  { current leader/candidate is active }
15   if  $leader = \perp$  and received at least two (OK,  $k$ ) messages
16   then  $leader \leftarrow k \bmod n$                         { now elect leader }
17   restart timer

18   upon  $timer > 2\delta$  do                                { timeout on current leader/candidate }
19   send (STOP,  $r$ ) to  $r \bmod n$                           { stop current leader/candidate }
20   StartRound( $r + 1$ )                                     { start next round }

21   upon receive (STOP,  $k$ ) with  $k \geq r$  do              { current leader abdicates leadership }
22   StartRound( $k + 1$ )                                     { start next round }

23   upon receive (OK,  $k$ ) or (START,  $k$ ) with  $k > r$  do
24   StartRound( $k$ )                                         { jump to round  $k$  }

```

**Fig. 2.** 3-stable algorithm for  $\Omega$  with reliable links.

---

## 6 Stable $\Omega$ with Message Losses

In our previous algorithm, we assumed that links do not lose messages, but in many systems this is not the case. We now modify our algorithms to deal with message losses. First note that if *all* messages can be lost, there is not much we can do, so we assume there is at least one eventually accessible process  $p$ . That means that  $p$  is correct and there is a time after which the links to and from  $p$  do not drop messages and are timely (see Sect. 2).

### 6.1 Expiring Links

So far, our model allowed links to deliver messages that have been sent long in the past. This behavior is undesirable because an out-of-date message can demote a leader that has been good recently. To solve this problem, we now use links that discard messages

older than  $\delta$  — we call them *expiring links*. Such links can be easily implemented from “plain” lossy links if processes have approximately synchronized, drift-free clocks, by using timestamps to expire old messages. We now make these ideas more precise.

**Informal definition.** Expiring links are lossy links that automatically drop old messages. To model such links, we change the property “No late delivery in good periods” of lossy links (Sect. 2) to require no late delivery *at all times*, not just when the link is good. More precisely, an expiring link  $p \rightarrow q$  satisfies the following properties:

- (No late delivery): If  $p$  sends  $m$  to  $q$  by time  $t - \delta$  then  $q$  does not receive  $m$  after  $t$ .
- (No creation or duplication): (same as before) Process  $q$  receives a message  $m$  from  $p$  at most once, and only if  $p$  previously sent  $m$  to  $q$ .
- (No loss in good periods): (same as before) If  $p$  sends  $m$  to  $q$  at time  $t - \delta$  and  $p \rightarrow q$  is good during times  $[t - \delta, t]$  then  $q$  eventually receives  $m$  from  $p$ .

**Implementation.** If processes have perfectly synchronized clocks, we can easily implement expiring links from plain lossy links as follows: (1) the sender timestamps  $m$  before sending it and (2) the receiver checks the timestamp and discards messages older than  $\delta$ . This idea also works when processes have  $\epsilon$ -synchronized, drift-free clocks, though the resulting link will have a  $\delta$  parameter that is  $2\epsilon$  larger than the  $\delta$  of the original links. It is also possible to expire messages even if clocks are not synchronized, provided they are drift-free or have a bounded drift, as we show in the full paper [ADGFT01]. Henceforth, we assume that all links are expiring links (this holds for all our  $\Omega$  algorithms that tolerate message losses).

## 6.2 $O(n)$ -Stable $\Omega$

Figure 3 shows an  $(n + 4)$ -stable algorithm for  $\Omega$  that works despite message losses. The algorithm is similar to our previous algorithm that assumes reliable links (Fig. 2), with only three differences: the first one is that in line 2,  $p$  sends the *START* message to all processes, not just to  $s \bmod n$ . The second difference is that there are no *STOP* messages. And the last difference is the addition of lines 21 and 22; without these two lines, the algorithm would not implement  $\Omega$  (it is not hard to construct a scenario in which the algorithm fails).

This new algorithm is  $O(n)$ -stable rather than  $O(1)$ -stable. To see why, consider the following scenario. Initially all processes are in round 0. At time  $2\delta + 1$  the following happens: (1) process 2 times out on round 0 and attempts to send  $(START, 1)$  to all, but crashes during the send and only sends to process 3 and (2) process 0 becomes accessible. At time  $3\delta + 1$ , process 3 receives  $(START, 1)$  and tries to send  $(START, 1)$  to all, but crashes and only sends to process 4. And so on. Then at time  $n\delta + 1$ , process 0 is the leader but it receives  $(START, 1)$  from process  $n - 1$  and demotes itself, even though it has been accessible during  $[2\delta + 1, n\delta + 1]$ .

**Theorem 3.** Consider a system with message losses (expiring links). Assume some process is eventually accessible. Figure 3 is an  $(n + 4)$ -stable communication-efficient algorithm for  $\Omega$ .

---

Code for each process  $p$ :

```

1  procedure StartRound( $s$ )                                { executed upon start of a new round }
2    if  $p \neq s \bmod n$  then send ( $START, s$ ) to all        { bring all to new round }
3     $r \leftarrow s$                                           { update current round }
4     $leader \leftarrow \perp$                                 { demote previous leader but do not elect leader quite yet }
5    restart timer

6  on initialization:
7    StartRound(0)
8    start tasks 0 and 1

9  task 0:                                                  { leader/candidate sends  $OK$  every  $\delta$  time }
10   loop forever
11     if  $p = r \bmod n$  and have not sent ( $OK, r$ ) within  $\delta$  then send ( $OK, r$ ) to all

12  task 1:
13   upon receive ( $OK, k$ ) with  $k = r$  do                  { current leader/candidate is active }
14     if  $leader = \perp$  and received at least two ( $OK, k$ ) messages
15       then  $leader \leftarrow k \bmod n$                     { now elect leader }
16       restart timer

17   upon timer  $> 2\delta$  do                                { timeout on current leader/candidate }
18     StartRound( $r + 1$ )                                  { start next round }

19   upon receive ( $OK, k$ ) or ( $START, k$ ) with  $k > r$  do
20     StartRound( $k$ )                                       { jump to round  $k$  }

21   upon receive ( $OK, k$ ) or ( $START, k$ ) from  $q$  with  $k < r$  do
22     send ( $START, r$ ) to  $q$                              { update process in old round }

```

**Fig. 3.**  $(n + 4)$ -stable algorithm for  $\Omega$  that tolerates message losses.

---

### 6.3 $O(1)$ -Stable $\Omega$

Our previous algorithm can tolerate message losses but it is only  $O(n)$ -stable. This can be troublesome if the number  $n$  of processes is large. We now provide a better algorithm that is 6-stable. We manage to get  $O(1)$ -stability by ensuring that a leader is not elected if there are long chains of messages that can demote the leader in the future.

Our algorithm is shown in Fig. 4. It is identical to our previous algorithm, except that there is a new ( $ALERT, k$ ) message. This message is sent to all when a process starts round  $k$ . When a process receives such a message from a higher round, the process temporarily sets its leader variable to  $\perp$  for  $6\delta$  time units. However, unlike with a  $START$  message, the process does not advance to the higher round.

**Theorem 4.** *Consider a system with message losses (expiring links). Assume some process is eventually accessible. Figure 4 is a 6-stable communication-efficient algorithm for  $\Omega$ .*



---

Code for each process  $p$ :

```

1  procedure StartRound( $s$ )                                { executed upon start of a new round }
2    send (ALERT,  $s$ ) to all
3    if  $p \neq s \bmod n$  then send (START,  $s$ ) to all          { bring all to new round }
4     $r \leftarrow s$                                            { update current round }
5     $leader \leftarrow \perp$                                    { demote previous leader but do not elect leader quite yet }
6    restart timer

7  on initialization:
8    StartRound(0)
9    start tasks 0 and 1

10 task 0:                                                    { leader/candidate sends OK every  $\delta$  time }
11   loop forever
12   if  $p = r \bmod n$  and have not sent (OK,  $r$ ) within  $\delta$  then send (OK,  $r$ ) to all

13 task 1:
14   upon receive (OK,  $k$ ) with  $k = r$  do                    { current leader/candidate is active }
15     if  $leader = \perp$  and received at least two (OK,  $k$ ) messages
16       and did not receive (ALERT,  $k'$ ) with  $k' > k$  within  $6\delta$ 
17     then  $leader \leftarrow k \bmod n$                           { now elect leader }
18     restart timer

19   upon timer  $> 2\delta$  do                                  { timeout on current leader/candidate }
20     StartRound( $r + 1$ )                                       { start next round }

21   upon receive (OK,  $k$ ) or (START,  $k$ ) with  $k > r$  do
22     StartRound( $k$ )                                           { jump to round  $k$  }

23   upon receive (OK,  $k$ ) or (START,  $k$ ) from  $q$  with  $k < r$  do
24     send (START,  $r$ ) to  $q$                                   { update process in old round }

25   upon receive (ALERT,  $k$ ) with  $k > r$  do
26      $leader \leftarrow \perp$                                    { suspend current leader }

```

**Fig. 4.** 6-stable algorithm for  $\Omega$  that tolerates message losses.

---

## 7 Stable $\Omega$ with Constant Election Time

In some applications, it is important to have a small *election time* — the time to elect a new leader when the system is leaderless. This time is inevitably large if there are crashes or slow links during the election. For instance, if an about-to-be leader crashes right before being elected, the election has to start over and the system will continue to be leaderless. Slow links often cause the same effect.

It is possible, however, to ensure small election time during *good periods* — periods with no slow links or additional crashes. In such periods, the election time of our previous algorithms is proportional to  $f$ , the number of crashes so far. This is because processes may go through  $f$  rounds trying to elect processes who are long dead. With a simple modification, however, it is possible to do much better and achieve a constant election

---

```

20      send (ALERT, r + 1) to all
20.1    send (PING, r) to all { ask who is alive }
20.2    wait for 2δ time or until receive (OK, k) or (START, k) with k > r
20.3    if received (OK, k) or (START, k) with k > r then return
20.4    S ← {q: received (PONG, r) from q} { responsive processes }
        { we assume that p responds to itself immediately, so S is never ∅ }
20.5    k ← smallest k' > r such that k' mod n ∈ S { round of next responsive process }
20.6    StartRound(k) { start next round }

20.7    upon receive (PING, k) from q do
20.8        send (PONG, k) to q

```

---

**Fig. 5.** Improving the election time in the algorithm of Fig. 4

---

time (independent of  $f$ ). The basic idea is that, when a process wants to start a new round, it first queries all processes to determine who is alive. Then, instead of starting the next round, the process skips all rounds of unresponsive processes. Using this idea, we can get a 6-stable algorithm with an election time of  $9\delta$ , as follows: we take the algorithm of Fig. 4 and replace its line 20 with the code shown in Fig. 5<sup>8</sup>

**Theorem 5.** *Consider a system with message losses (expiring links). Assume some process is eventually accessible. If we replace line 20 in Fig. 4 with the code in Fig. 5 we obtain a 6-stable communication-efficient algorithm for  $\Omega$ . Its election time is  $9\delta$  when there are no slow links or additional crashes.*

## 8 Leader Election with View Numbers

It may be useful to tag leaders with a *view number* such that there is at most one leader per view number and eventually processes agree on the view number of the leader. More precisely, we define a variant of  $\Omega$ , which we call  $\Omega^+$ , in which each process outputs a pair  $(p, v)$  or  $\perp$ , where  $p$  is a process and  $v$  is a number.  $\Omega^+$  guarantees that (1) if some process outputs  $(p, v)$  and some process outputs  $(q, v)$  then  $p = q$  and (2) there exists a correct process  $\ell$ , a number  $v_\ell$  and a time after which, for every alive process  $p$ ,  $p$  outputs  $(\ell, v_\ell)$ .

It turns out that our  $\Omega$  algorithms can be made to output a view number with no modifications: they can simply output the current round  $r$ . By doing that, it is not hard to verify that our algorithms actually implement  $\Omega^+$ .

## 9 An Efficient Algorithm for $\Diamond P$

Recall that, at each alive process  $p$ , the eventually perfect failure detector  $\Diamond P$  outputs a set of trusted processes, such that there is a time after which the set of  $p$  contains

---

<sup>8</sup> The same idea can be applied to get a constant election with our other algorithms.

---

```

Code for each process  $p$ :
1  procedure  $StartRound(s)$                                      { executed upon start of a new round }
2    if  $p \neq s \bmod n$  then send ( $START, s$ ) to all
3     $r \leftarrow s$                                              { update current round }
4     $leader \leftarrow s \bmod n$ 
5     $trust \leftarrow \Pi$                                        { trust all initially }
6    restart timer

7  on initialization:
8     $StartRound(0)$ 
9    start tasks 0 and 1

10 task 0:                                                       { leader updates  $trust$  and sends  $OK$  every  $\delta$  time }
11   loop forever
12     if  $p = r \bmod n$  and have not sent ( $OK, r, *$ ) within  $\delta$  time then
13       if  $p$  has been in round  $r$  for at least  $2\delta$  time then
14         for each  $q \in trust$  s.t.  $p$  did not receive ( $ACK, r$ ) from  $q$  in the last  $2\delta$  time do  $trust \leftarrow trust \setminus \{q\}$ 
15         send ( $OK, r, trust$ ) to all

16 task 1:
17   upon receive ( $OK, k, tr$ ) with  $k = r$  do                     { current leader is active }
18     if  $p \notin tr$  then  $StartRound(r + 1)$                    { leader does not trust  $p$ , so  $p$  starts new round }
19     else
20        $trust \leftarrow tr$ 
21       send ( $ACK, r$ ) to  $r \bmod n$ 
22       restart timer

23   upon timer  $> 2\delta$  do                                     { timeout on current leader }
24      $StartRound(r + 1)$                                        { start next round }

25   upon receive ( $OK, k, tr$ ) or ( $START, k$ ) with  $k > r$  do
26     if received ( $OK, *, *$ ) then send ( $ACK, k$ ) to  $k \bmod n$ 
27      $StartRound(k)$                                            { jump to round  $k$  }

28   upon receive ( $OK, k, tr$ ) or ( $START, k$ ) from  $q$  with  $k < r$  do
29     send ( $START, r$ ) to  $q$ 

```

---

**Fig. 6.** An efficient algorithm for  $\Diamond P$ .

process  $q$  if and only if  $q$  is correct. We now give an algorithm for  $\Diamond P$  that is both robust and efficient: In contrast to previous implementations of  $\Diamond P$ , our algorithm works in systems where only  $n$  bidirectional links are required to be eventually timely, and there is a time after which only  $n$  bidirectional links carry messages.

Our algorithm, shown in Fig. 6, tolerates message losses with expiring links. It is based on the algorithm in Fig. 3, and the difference is that (1) there are no *ALERT* messages and (2) there is a mechanism to get the list *trust* of trusted processes of  $\Diamond P$ : When processes receive *OK*, they send *ACK* to the leader, and the leader sets *trust* to the set of processes from which it received *ACK* recently. The leader then sends its *trust* to other processes, by piggybacking it in the *OK* messages. Upon receiving *OK*, a process  $q$  checks if the leader's *trust* contains  $q$ . If so, the process sets its own *trust* to the leader's. Else, the process notices that the leader has made a mistake, and so it starts the next round.

We assume that if a process sends a message to itself, that message is received and processed immediately.

**Theorem 6.** Consider a system with message losses (lossy links). Assume some process is eventually accessible. Figure 6 is an algorithm for  $\diamond P$ . With this algorithm, there is a time after which only  $n$  bidirectional links carry messages.

## References

- [ACT99] M. K. Aguilera, W. Chen, and S. Toueg. Using the heartbeat failure detector for quiescent reliable communication and consensus in partitionable networks. *Theoretical Computer Science*, 220(1):3–30, June 1999.
- [ACT00] M. K. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. *Distributed Computing*, 13(2):99–125, April 2000.
- [ADGFT01] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Stable leader election. Technical Report 2001/04, LIAFA, Université D. Diderot, 2 Place Jussieu, 75251, Paris Cedex 05, France, 2001.
- [CHT96] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, July 1996.
- [Chu98] F. Chu. Reducing  $\Omega$  to  $\diamond W$ . *Information Processing Letters*, 67(6):298–293, September 1998.
- [CT96] T. D. Chandra and S. Toueg. Unreliable failure detectors for asynchronous systems. *Journal of the ACM*, 43(2):225–267, March 1996.
- [CTA00] W. Chen, S. Toueg, and M. K. Aguilera. On the quality of service of failure detectors. In *International Conference on Dependable Systems and Networks (DSN 2000)*, pages 191–200, New York, June 2000. A full version of this paper will appear in the IEEE Transactions on Computers.
- [DHW98] C. Dwork, J.Y. Halpern, and O. Waarts. Performing work efficiently in the presence of faults. *SIAM Journal on Computing*, 27(5):1457–1491, 1998.
- [DPLL97] R. De Prisco, B. Lampsom, and N. Lynch. Revisiting the Paxos algorithm. In *Proceedings of the 11th Workshop on Distributed Algorithms(WDAG)*, pages 11–125, Saarbrücken, September 1997.
- [FRT00] C. Fetzer, M. Raynal, and F. Tronel. A failure detection protocol based on a lazy approach. Research Report 1367, IRISA, November 2000.
- [LAF99] M. Larrea, S. Arévalo, and A. Fernández. Efficient algorithms to implement unreliable failure detectors in partially synchronous systems. In *Proceedings of the 13th International Symposium on Distributed Algorithms(DISC99)*, pages 34–48, Bratislava, September 1999.
- [Lam98] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.
- [Lar00] M. Larrea, November 2000. Personal communication.
- [LFA00a] M. Larrea, A. Fernández, and S. Arévalo. Eventually consistent failure detectors. In *Brief Annoucement the 14th International Symposium on Distributed Algorithms(DISC00)*, Toledo, October 2000.
- [LFA00b] M. Larrea, A. Fernández, and S. Arévalo. Optimal implementation of the weakest failure detector for solving consensus. In *in Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems, SRDS 2000*, pages 52–59, Nuremberg, Germany, October 2000.
- [MR00] A. Mostefaoui and M. Raynal. Leader-based consensus. Research Report 1372, IRISA, December 2000.
- [VRMMH98] R. Van Renesse, Y. Minsky, and M. M. Hayden. A gossip-based failure detection service. In *Proceedings of Middleware '98 (Sept. 1998)*, September 1998.

# Adaptive Long-Lived $O(k^2)$ -Renaming with $O(k^2)$ Steps<sup>\*</sup>

Michiko Inoue<sup>1</sup>, Shinya Umetani<sup>1</sup>, Toshimitsu Masuzawa<sup>2</sup>, and Hideo Fujiwara<sup>1</sup>

<sup>1</sup> Graduate School of Information Science,  
Nara Institute of Science and Technology (NAIST)  
Ikoma Nara, 630-0101 JAPAN

<sup>2</sup> Graduate School of Engineering Science, Osaka University  
Toyonaka Osaka, 560-8531 JAPAN

**Abstract.** We present a long-lived renaming algorithm in the read/write shared memory model. Our algorithm is adaptive to the point contention  $k$  and works with bounded memory and bounded values. We consider the renaming problem where each process obtains the new name in the range  $1, \dots, k(2k - 1)$ . In this paper, we present an algorithm with  $O(k^2)$  step complexity and  $O(n^2 N)$  space complexity, where  $n$  and  $N$  are an upper bound of  $k$  and the number of processes, respectively. The previous best result under the same problem setting is the algorithm with  $O(k^3)$  step complexity and  $O(n^3 N)$  space complexity presented by Afek et. al [1]. They also presented the algorithm with  $O(k^2 \log k)$  step complexity and  $O(n^3 N)$  space complexity under the condition where unbounded values are allowed. That is, we improve the above two algorithms.

## 1 Introduction

We consider a long-lived  $M$ -renaming problem in an asynchronous read/write shared memory model. In the problem, every process repeatedly acquires a new name in the range  $\{1, 2, \dots, M\}$ , and releases it after the use. The problem requires that no two processes keep the same name concurrently. The renaming algorithm is very useful in the situation where the number of active processes is much smaller than the number of total processes which have a potential for participation. Since the complexities of most distributed algorithms depend on the name space of processes, we can reduce the complexities by reducing the name space using the renaming algorithm. For this purpose, long-lived renaming algorithms are used in other long-lived applications such as *atomic snapshot* [2], *immediate snapshot* [2], or *collet* [3].

Renaming algorithms are required to have a small name space and low complexity. In the first renaming algorithms [4, 5], their step complexities depend on the number  $N$  of the processes which have a potential for participation in the algorithms and each process is allowed to acquire the new name only once (*one-time renaming*). An algorithm is called to be *fast* if its step complexity depends on not  $N$  but the upper bound  $n$  of the number of the processes which actually participate in the algorithm. One-time fast renaming

---

<sup>\*</sup> This work is supported in part by Japan Society for the Promotion of Science (JSPS), Grant-in-Aid for Scientific Research C(2) (No.12680349) and A (No.12780225).

**Table 1.** Adaptive long-lived renaming algorithms.

Step complexity	Name-space	Space complexity	Contention	Value Size	Reference
$O(k^2)$	$4k^2$	$O(n^2 N)$	interval	bounded	[11]
$O(k^2 \log k)$	$2k^2 - k$	unbounded	point	bounded	[11]
$O(k^2 \log k)$	$2k^2 - k$	$O(n^3 N)$	point	unbounded	[11]
$O(k^3)$	$2k^2 - k$	$O(n^3 N)$	point	bounded	[11]
$Exp(k)$	$2k - 1$	$O(n^3 N)$	point	unbounded	[11]
$O(k^4)$	$2k - 1$	unbounded	point	unbounded	[12]
$O(k^2)$	$2k^2 - k$	$O(n^2 N)$	point	bounded	this paper

algorithms [6] and *long-lived* fast renaming algorithms [7,8] have been proposed. Fast renaming algorithms use the upper bound  $n$  in the algorithms and it needs to be known in advance.

Recently, renaming algorithms where the step complexities depend on only the number of the processes which actually participate in the algorithm were proposed. Such algorithms are said to be *adaptive*. In adaptive algorithms, the number of actually active processes is unknown in advance. Table 1 shows the results on adaptive long-lived renaming algorithms. Attiya et al. [9] and Afek et al. [10] proposed one-time renaming algorithms which step complexity are functions of the interval contention, where the interval contention is the the number of active processes in the execution interval. Afek et al. [11] first proposed adaptive long-lived renaming, where some algorithms are adaptive to the interval contention and use bounded memory and others are adaptive to the *point contention* and use unbounded memory. The point contention  $k$  is the the maximum number of processes executing concurrently at some point. Afek et al. [11] improved their previous algorithms [11]. They proposed long-lived renaming algorithms adaptive to the point contention. Their algorithms are a  $(2k^2 - k)$ -renaming algorithm with  $O(k^2 \log k)$  step complexity and  $O(n^3 N)$  space complexity using unbounded values, a  $(2k^2 - k)$ -renaming algorithm with  $O(k^3)$  step complexity and  $O(n^3 N)$  space complexity using bounded values, and linear name-space renaming algorithms with exponential step complexity and  $O(n^3 N)$  space complexity. Attiya et al. [12] proposed point-contention-adaptive long-lived  $(2k - 1)$ -renaming algorithm with  $O(k^4)$  step complexity.

In this paper, we present a point-contention-adaptive long-lived  $(2k^2 - k)$ -renaming algorithm with  $O(k^2)$  step complexity and  $O(n^2 N)$  space complexity using bounded values. That is, our algorithm improves the previous two results of  $(2k^2 - k)$ -renaming algorithms [11]. Our algorithm is based on the  $(2k^2 - k)$ -renaming algorithm with  $O(k^3)$  step complexity [11]. In the algorithm [11], every process which wants to get a new name visits series of *sieves* and tries to win in some sieve. In each sieve, a procedure *latticeAgreement* [9] is used to obtain a snapshot of participants in the sieve. We replace this procedure with a simple procedure, and can reduce both the step complexity and the space complexity. The replaced procedure is based on an adaptive collect algorithm presented in [13].

This paper is organized as follows. We give some definitions in Section 2, and briefly describe the previous  $(2k^2 - k)$ -renaming algorithm [1] in Section 3. Sections 4 and 5 show our efficient  $(2k^2 - k)$ -renaming algorithm and its correctness. Finally, we conclude this paper in Section 6.

## 2 Definition

Our computation model is an asynchronous read/write shard memory model [4]. A shared memory model consists of a set of  $N$  asynchronous processes  $p_0, p_1, \dots, p_{N-1}$  and a set of registers shared by the processes. Each process has a unique identifier, and we consider that an identifier of  $p_i$  is  $i$ . The processes communicate each other only through the registers which provide two atomic operations write and read. We assume *multi-writer multi-reader* registers, that is, any process can write to and read from any registers.

The definition of a *long-lived renaming problem* and a *point contention* are the same as the related works [1].

In the long-lived  $M$ -renaming problem, process repeatedly acquires and releases names in the range  $\{1, 2, \dots, M\}$ . A renaming algorithm provides two procedures  $\text{getName}_i$  and  $\text{releaseName}_i$  for each process  $p_i$ . A process  $p_i$  uses the procedure  $\text{getName}_i$  to get a new name, and uses the procedure  $\text{releaseName}_i$  to release it. Each process  $p_i$  alternates between invoking  $\text{getName}_i$  and  $\text{releaseName}_i$ , starting with  $\text{getName}_i$ .

An execution of an algorithm is a (possibly infinite) sequence of register operations and invocations and returns of procedures where each process follows the algorithm. Let  $\alpha$  be an execution of a long-lived renaming algorithm, and let  $\alpha'$  be some finite prefix of  $\alpha$ . Process  $p_i$  is *active* at the end of  $\alpha'$  if  $\alpha'$  includes an invocation of  $\text{getName}_i$  which does not precede any return of  $\text{releaseName}_i$  in  $\alpha'$ . Process  $p_i$  *holds* a name  $y$  at the end of  $\alpha'$  if the last invocation of  $\text{getName}_i$  returned  $y$  and  $p_i$  has been active ever since the return. A long-lived renaming algorithm should guarantee the following *uniqueness*: If active processes  $p_i$  and  $p_j$  ( $i \neq j$ ) hold names  $y_i$  and  $y_j$ , respectively, at the end of some finite prefix of some execution, then  $y_i \neq y_j$ .

The *contention* at the end of  $\alpha'$ , denoted  $\text{Cont}(\alpha')$ , is the number of active processes at the end of  $\alpha'$ . Let  $\beta$  be a finite interval of  $\alpha$ , that is,  $\alpha = \alpha_1\beta\alpha_2$  for some  $\alpha_1$  and  $\alpha_2$ . The *point contention* during  $\beta$ , denoted  $\text{PntCont}(\beta)$ , is the maximum contention in prefixes  $\alpha_1\beta'$  of  $\alpha_1\beta$ .

A renaming algorithm has *adaptive name space* if there is a function  $f$ , such that the name obtained in an interval of  $\text{getName}$ ,  $\beta$ , is in the range  $\{1, \dots, f(\text{PntCont}(\beta))\}$ . Step complexity of a renaming algorithm is the worst case number of steps performed by some  $p_i$  in an interval  $\beta$  of  $\text{getName}_i$  and in the following  $\text{releaseName}_i$ . A renaming algorithm has *adaptive step complexity* if there is a bounded function  $S$ , such that the number of steps performed by  $p_i$  in any interval  $\beta$  of  $\text{getName}_i$  and in the following  $\text{releaseName}_i$  is at most  $S(\text{PntCont}(\beta))$ . Space complexity of a renaming algorithm is the number of registers used in the algorithm.

### 3 $O(k^3)$ -Algorithm

Our renaming algorithm is based on the  $(2k^2 - k)$ -renaming algorithm presented in [1], which is a long-lived renaming algorithm adaptive to a point contention  $k$  with  $O(k^3)$  step complexity using bounded memory and bounded values. We call this  $O(k^3)$ -algorithm. To present our algorithm, we briefly describe the  $O(k^3)$ -algorithm. Figure 1 shows top level procedures `getName` and `releaseName` in the  $O(k^2)$ -algorithm that we will present later. In this paper, we show algorithms in Fig. 12, 13, and 14 where all of them are pseudo codes for process  $p_i$ .

These pseudo codes in Fig. 1 are the same as the  $O(k^3)$ -algorithm except that a procedure `interleaved_sc_sieve` returns a set of pairs  $\langle id, sp \rangle$ . In the  $O(k^3)$ -algorithm, this procedure returns a set of identifiers that are the first components of  $\langle id, sp \rangle$ . In our  $O(k^2)$ -algorithm, to reduce both step and space complexities, we use different data structures from the  $O(k^3)$ -algorithm inside lower level procedures mentioned later. The second component  $sp$  of  $\langle id, sp \rangle$  is a pointer to the data structure where some information about a process  $p_{id}$  is stored. In the  $O(k^3)$ -algorithm, the value  $id$  is sufficient to point the corresponding data structure, while our proposed algorithm needs such a pointer like  $sp$  for efficient data space management. However,  $sp$  itself is used only in the procedures called in a procedure `releaseName`, and is not used the top level procedures. This means the top level two procedures `getName` and `releaseName` behave in the same way between the  $O(k^3)$ -algorithm and our  $O(k^2)$ -algorithm.

The difference between two algorithms are details of procedures `interleaved_sc_sieve`, `leave` and `clear` called in the top level procedures. Each process gets a new name and releases it as follows.

**getting a name:** A process visits *sieves*  $1, 2, \dots$  until it wins in some sieve. Each sieve has  $2N$  copies, where one copy is work space for processes which visit the sieve concurrently. A process  $p_i$  visiting a sieve  $s$  enters one copy and obtains a set of process identifiers if it satisfies some conditions. If  $p_i$  gets a non-empty set  $W$  of process identifiers including its identifier  $i$ ,  $p_i$  wins in the sieve. If  $p_i$  wins in  $s$ ,  $p_i$  gets a name  $\langle s$ , the rank of  $p_i$  in  $W \rangle$ .

**releasing a name:** A process  $p_i$  leaves the copy from which  $p_i$  got a name to show that  $p_i$  released the name.

Each sieve has  $2N$  copies  $0, 1, \dots, 2N - 1$ . Each process uses a copy in a sieve  $s$  designated by a variable `sieve[s].count`. The first component of the variable changed to  $0, 1, \dots, 2N - 1, 0, 1, \dots$  cyclically. We can associate a *round* with the value of `sieve[s].count` which means how many times the variable is updated to the current value. If a process sees `sieve[s].count` with a round  $r$ , we say that the process uses the designated copy in the round  $r$ . The  $O(k^3)$ -algorithm guarantees the following.

- The processes which enter the same copy in the same round get the identical non-empty set of process identifiers or an empty set. Moreover, Processes enter a copy after all names assigned from the previous copy are released. These imply the uniqueness of concurrently assigned names.
- If at least one process enter some copy in some round, at least one process wins. This puts bounds to the number of sieves each process visits.



---

Shared variables :

```

sieve[1, ..., 2n - 1] {
  count : (integer, Boolean), initially (0, 0);
  status[0, ..., N - 1] : Boolean, initially false;
  // 2N copies; each copy consists of inside, allDone and list.
  inside[0, ..., 2N - 1] : Boolean, initially false;
  allDone[0, ..., 2N - 1] : Boolean, initially false;
  // list is only for  $O(k^2)$ -algorithm.
  list[0, ..., 2N - 1] {
    mark[0, ..., n - 1] : Boolean, initially false;
    view[0, ..., n - 1] : set of (id, integer), initially  $\perp$ ;
    id[0, ..., n - 1] : id, initially  $\perp$ ;
    X[0, ..., n - 1] : integer, initially  $\perp$ ;
    Y[0, ..., n - 1] : Boolean, initially false;
    done[0, ..., n - 1] : Boolean, initially false;
  }
}

```

Non-shared Global variables :

```

nextC, c : integer, initially 0; nextDB, dirtyB : Boolean, initially 0;
sp : integer, initially  $\perp$ ;
W : view (set of (id, integer)), initially  $\emptyset$ ; s : integer, initially 0;

```

procedure getName()

```

1  s = 0;
2  while (true) do
3    s++;
4    sieve[s].status[i] = active;
5    (c, dirtyB) = sieve[s].count;
6    nextC = c + 1 mod 2N;
7    if (nextC = 0) then nextDB = not dirtyB;
8    else nextDB = dirtyB;
9    if ((nextC mod N = i) or (sieve[s].status[nextC mod N] = idle)) then
10     W = interleaved_sc_sieve(sieve[s], nextC, nextDB);
11     if ((i, sp) ∈ W for some sp) then
12       sieve[s].count = (nextC, nextDB);
13       return (s, rank of i in W);
14     else-if (sieve[s].allDone[nextC] = nextDB) then clear(sieve, nextC);
15     sieve[s].status[i] = idle;
16  od;

```

procedure releaseName()

```

17  leave(sieve[s], nextC, nextDB);
18  if (sieve[s].allDone[nextC] = nextDB) then
19    clear(sieve, nextC);
20  sieve[s].status[i] = idle;

```

---

**Fig. 1.** Outline of  $O(k^3)$ -algorithm and  $O(k^2)$ -algorithm.

- When a process  $p_i$  is the inside of a sieve  $s$ , no other process can enter the copies  $i$  and  $i + N$ . This puts bounds to the number of copies used concurrently in one sieve, and guarantees that no copy is used concurrently in the different rounds.

Each copy is initialized to be reused in the next round. When a process  $p_i$  is leaving the copy  $c$  of a sieve  $s$  (lines 14 and 18),  $p_i$  initializes  $c$  if all the names assigned from  $c$  are released. A Boolean variable  $sieve[s].allDone[c]$  is used as a signal that a round in the copy has been finished. The value  $nextDB$  differs with the parity of the round. (lines 7 and 8).

In the  $O(k^3)$ -algorithm, each process can win in some sieve among the first  $(2k - 1)$  sieves, and enters at most one copy in each sieve. In each copy, processes try to get some non-empty set of process identifiers by using procedures **latticeAgreement** and **candidates**, where **latticeAgreement** returns a snapshot of participants of the procedure and **candidates** returns the minimum snapshot among the snapshots obtained by **latticeAgreement**. In the  $O(k^3)$ -algorithm, in each copy, each process executes **latticeAgreement** once with  $O(k \log k)$  steps, and executes a procedure **clear** at most once to initialize the copy with  $O(k^2)$  steps. Therefore, the step complexity of the  $O(k^3)$ -algorithm is  $O(k^3)$ .

The space complexity is as follows. The algorithm uses  $2n - 1$  sieves, each of which has  $2N$  copies. Each copy has  $O(n^2)$  registers for **latticeAgreement** which dominates the space complexity of a copy. Therefore, the total space complexity is  $O(n^3N)$ . Since each process gets a new name  $\langle sieve, \text{its rank in } W \rangle$  where  $W$  is a non-empty set obtained in the sieve where the process wins, the name space is  $(2k - 1)k = 2k^2 - k$ .

## 4 $O(k^2)$ -Algorithm

We replace the **latticeAgreement** and the **candidates** in the  $O(k^3)$ -algorithm. Especially, we replace the **latticeAgreement** with a simple procedure, and achieve  $O(k^2)$  step complexity and  $O(n^2N)$  space complexity while using bounded values. We call our algorithm  $O(k^2)$ -algorithm. Figures 2 and 3 show procedures used in the  $O(k^2)$ -algorithm.

In the  $O(k^3)$ -algorithm, processes which concurrently enter the same copy obtain the identical set  $W$  if they obtain non-empty set. To achieve this, procedures **latticeAgreement** and **candidates** are used. Each process entering a copy invokes **latticeAgreement** to capture a snapshot (i.e., the values at some point) of processes which have entered the same copy in the same round. Then the process invokes **candidates** to find the minimum snapshot among the snapshots obtained by **latticeAgreement**. The procedure **candidates** can return the minimum snapshot only when the minimum snapshot can be identified, where “minimum” means the minimum one among snapshots obtained in the same copy in the same round including snapshots obtained by other processes later. Since the minimum snapshot is unique, some processes can obtain the identical non-empty set  $W$ . If  $k$  processes execute **latticeAgreement** concurrently,  $O(k^2)$  registers are used. To initialize these register, some process takes  $O(k^2)$  steps in one sieve. This dominates the step complexity of the  $O(k^3)$ -algorithm.

In our  $O(k^2)$ -algorithm, some processes obtain a snapshot of processes registered in the copy (procedure **partial\_scan**). This can be achieved by invoking **collect** twice.

Then, processes find the minimum snapshot by invoking **candidates** if the minimum snapshot can be identified. Though our algorithm is simple, it guarantees that at least one processes obtain the non-empty identical set. Moreover, our procedures use  $O(k)$  registers if  $k$  processes enter the same copy in the same round. Therefore,  $O(k)$  steps are sufficient to initialize the copy. This reduces the step complexity.

We explain a procedure **interleaved\_sc\_sieve** (Fig 2). A process *enters* a copy  $c$ , if all the names assigned from the previous copy are released and the current copy is free (line 30). The process can notice that all the names assigned from the previous copy  $(c - 1 \bmod 2N)$  are released by checking a variable  $sieve.allDone[c - 1 \bmod 2N]$ . The Boolean variable  $sieve.allDone[c]$  changes after all names assigned from a copy  $c$  in a sieve  $sieve$  are released.

If a process obtains a non-empty set  $W$  of process identifiers,  $W$  is a set of candidates of winners in this sieve. After the all candidates left this copy, the copy is initialized to be reused in the next round (**clear**). However, some slow process excluding  $W$  may still work in the copy after the initialization started. Therefore, after every operation to shared registers, each process checks whether the copy has been finished or not. If the process notices that the copy has been finished, it initializes the last modified register and leaves the sieve. This mechanism is implemented by *interleave* (line 22) which is the same as the  $O(k^3)$ -algorithm.

A process entering a copy  $c$  scans  $c$ . To scan a copy, our algorithm uses *total-contention-adaptive collect* and **register** with  $O(k)$  step complexity presented in [13], where the total contention is the number of active processes in an execution of the algorithm. In the  $O(k^2)$ -algorithm, these two procedures are executed by only the processes which enter the same copy concurrently. They all read **false** from  $sieve.inside[nextC]$  and then write **true** to it (lines 30 and 31). This means they are concurrently active in a point immediately after they all read the value **false** from  $sieve.inside[nextC]$  and before any process writes the value **true** to it. Therefore, we can use these procedures as point contention adaptive procedures.

To scan a copy  $c$ , processes try to register to  $c$  at first. In the **collect** presented in [13], every process can register to a *collect tree*. The collect tree is a binary tree where each node is a splitter presented in [6]. A process which enters a splitter exits with either **stop**, **left** or **right**. It guarantees that, among  $l$  processes which enter the same splitter, (1) at most one process obtains **stop**, (2) at most  $l - 1$  processes obtain **left**, and (3) at most  $l - 1$  processes obtain **right**. Since it is enough to get non-empty set of process identifiers, we do not need for all processes to register. Therefore, we simplify the **register** presented in [13]. We use only one direction of a splitter, and use a *collect list* in stead of the collect tree. The modified splitter returns **stop**, **next** or **abort** in stead of **stop**, **left** or **right**, respectively. The property of the splitter implies that (1) at least one process register at some node in a collect list, and (2) at most one process register at each splitter. Figure 4 shows procedures **register**, **collect** and **splitter**, and Figure 5 shows a collect list..

In the procedure **collect**, a process gets a set of process identifiers which are registered. We call such a set a *view*. Our **collect** is the same as **collect**[13] except that we search in the list. In **collect**, a process just searches the list from its root until it reaches an unmarked splitter. The **collect** returns a view consisting of all process identifiers which registered before invoking the **collect** and some process identifiers which register

---

Non-shared Global variables :

*last\_modified* : points to last shared variable modified by  $p_i$ ;  
                   // *last\_modified* is assumed to be updated  
                   // immediately before the write.

*mysplitter* : integer, initially  $\perp$ ;

```

procedure interleaved_sc_sieve(sieve, nextC, nextDB)
    // interleave is a two part construct.
    // Part I of the interleave is executed
    // after every read or write to a shared variable in Part II,
    // the sc.sieve() and any procedure recursively called from sc.sieve().
21  last_modified =  $\perp$ ;
22  interleave { // Part I
23      if (sieve.allDone[nextC] = nextDB) then
24          if (last_modified  $\neq$   $\perp$ ) then
25              write initial value to last_modified;
26          return  $\emptyset$ ;          // abort current sc.sieve(), s, and continue to next sieve.
27  }{ // Part II
28      return sc.sieve(sieve, nextC, nextDB);
29  }
```

```

procedure sc.sieve(sieve, nextC, nextDB)
30  if (previousFinish(sieve, nextC, nextDB) and sieve.inside[nextC] = false) then
31      sieve.inside[nextC] = true;
32      mysplitter = register(sieve.list[nextC]);
33      if (mysplitter  $\neq$   $\perp$ ) then
34          sieve.list[nextC].view[mysplitter] = partial.scan(sieve.list[nextC]);
35          W = candidates(sieve, nextC);
36          if ( $\langle p_i, \textit{mysplitter} \rangle \in W$ ) then return W;
37          sieve.list[nextC].done[mysplitter] = true;
38          W = candidates(sieve, nextC);
39          leave(sieve, nextC, nextDB);
40  return  $\emptyset$ ;
```

```

procedure previousFinish(sieve, nextC, nextDB)
41  if (nextC  $\neq$  0 and sieve.allDone[nextC - 1 mod 2N] = nextDB)
42      then return true;
43  if (nextC = 0 and sieve.allDone[2N - 1]  $\neq$  nextDB)
44      then return true;
45  return false;
```

---

**Fig. 2.** Procedures of  $O(k^2)$ -algorithm: part I.

concurrently with the execution of **collect**. Let  $V$  be a view obtained by an execution of **collect**, and  $V_1$  and  $V_2$  be sets of process identifiers which has registered immediately before and after the execution, respectively. The **register** and **collect** guarantee

---

```

procedure partial_scan(list)
46   $V_1 = \text{collect}(\textit{list});$ 
47   $V_2 = \text{collect}(\textit{list});$ 
48  if ( $V_1 = V_2$ ) then return  $V_1$ ;
49  else return  $\emptyset$ ;

procedure candidates(sieve, copy)
50   $sp = 0$ ;  $V = \emptyset$ ;
51  while (sieve.list[copy].mark[sp] = true) do
52    if (sieve.list[copy].view[sp]  $\neq \perp$ ) then
53       $V = V \cup \{\textit{sieve.list}[\textit{copy}].\textit{view}[\textit{sp}]\}$ ;
54       $sp++$ ;
55  od;
56  if  $V = \emptyset$  then return  $\emptyset$ ;
57   $U = \min\{\textit{view} \mid \textit{view} \in V \text{ and } \textit{view} \neq \emptyset\}$ ;
58  if  $U \neq \emptyset$  and for every  $\langle j, sp \rangle \in U$ , sieve.list[copy].view[sp]  $\supseteq U$ 
    or sieve.list[copy].view[sp] =  $\emptyset$  then
59    return  $U$ ;
60  else
61    return  $\emptyset$ ;

procedure clear(sieve, nextC)
62  sieve.inside[nextC] = false;
63   $sp = 0$ ;
64  while (sieve.list[nextC].mark[sp] = true) do
65    write initial value to a splitter sp in sieve.list[nextC];
66     $sp++$ ;
67  od;

procedure leave(sieve, nextC, nextDB)
68  sieve.list[nextC].done[mysplitter] = true;
69  if  $W \neq \emptyset$  and for every  $\langle j, sp \rangle \in W$ , sieve.list[nextC].done[sp] = true then
70    sieve.allDone[nextC] = nextDB;

```

---

**Fig. 3.** Procedures of  $O(k^2)$ -algorithm: part II.

$V_1 \subseteq V \subseteq V_2$ . This implies that if a process obtains the identical set by consecutive two invocations of **collect**, the set is a snapshot of processes which have been registered at some point between two **collects**. A procedure **candidates** returns the minimum snapshot. If a process obtains a view including its identifier by the **candidates**, the process becomes a winner.

## 5 Correctness of $O(k^2)$ -Algorithm

We briefly show the correctness of our algorithm. Since the outline of the  $O(k^2)$ -algorithm is the same as the  $O(k^3)$ -algorithm, it is enough to show the procedure

---

```

procedure register(list)
71  sp = 0;
72  while (true)
73    list.mark[sp] = true;
74    move = splitter(list, sp);
75    if (move = next) then
76      sp++;
77    if (move = abort) then
78      return  $\perp$ ;
79    if (move = stop) then
80      list.id[sp] = i;
81      return sp;
82  od;

procedure collect(list)
83  sp = 0; V =  $\emptyset$ ;
84  while (list.mark[sp] = true)
85    if (list.id[sp]  $\neq \perp$ ) then V = V  $\cup$  { (list.id[sp], sp) }
86    sp++;
87  od;
88  return V;

procedure splitter(list, currentsp)
89  list.X[currentsp] = i;
90  if (list.Y[currentsp] = true) then return abort;
91  list.Y[currentsp] = true;
92  if (list.X[currentsp] = i) then return stop;
93  else return next;

```

---

Fig. 4. Collect.

`partial_scan` and `candidates` in the  $O(k^2)$ -algorithm work well on behalf of `latticeAgreement` and `candidates` in the  $O(k^3)$ -algorithm. We show that processes which enter the same copy in the same round get the identical non-empty set of process identifiers or an empty set and show that at least one process enter some copy in some round, at least one process wins (obtains a set including its identifier).

We show some basic properties for `partial_scan` and `candidates`. By the same access control as the  $O(k^3)$ -algorithm, our  $O(k^2)$ -algorithm guarantees that all processes entering a copy in some round leave and the copy is initialized before the copy is used in the next round. Therefore, the behavior of some copy in some round is independent of the behavior of the previous rounds in the copy. The following lemmas concern a copy in one round.

**Lemma 1.** *If  $W_1$  and  $W_2$  are non-empty set returned by invocations of `candidate` for the same copy  $c$  in the same round of the same sieve  $s$  then  $W_1 = W_2$ .*

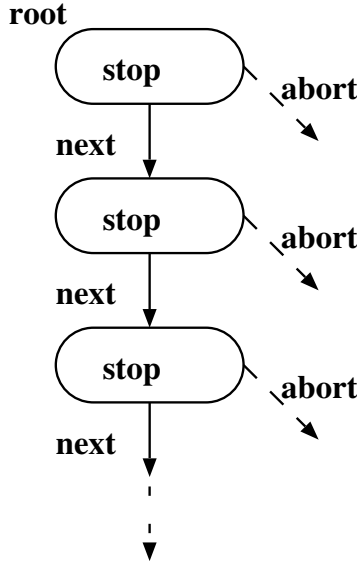


Fig. 5. Collect list.

*Proof.* We prove this lemma by contradiction. Assume  $W_1 \neq W_2$ . Since  $W_1$  and  $W_2$  are snapshots of processes which have registered,  $W_1 \subset W_2$  or  $W_2 \subset W_1$  holds. We assume  $W_1 \subset W_2$  without loss of generality. A snapshot returned by **candidate** is a snapshot obtained by some process using **partial\_scan** in the same copy in the same round. Let  $p_j$  be a process which obtains a snapshot  $W_1$  by **partial\_scan**, and  $p_m$  be a process which obtains  $W_2$  by **candidates**. Since  $p_j$  searches a collect list after it registered at some splitter  $sp$ ,  $p_j \in W_1$  holds. Since  $p_j$  is the only process which updates the variable  $s.list[c].view[sp]$  in this round, the value of  $s.list[c].view[sp]$  must be the initial value  $\perp$  or  $W_1$ . However,  $p_m$  sees that  $s.list[c].view[sp] \supseteq W_2$  or  $s.list[c].view[sp] = \emptyset$ . A contradiction.

By Lemma 1 and the fact that each copy is used after all the names assigned from the previous copy are released, we can show the following uniqueness.

**Lemma 2.** *If active processes  $p_i$  and  $p_j$  ( $i \neq j$ ) hold names  $y_i$  and  $y_j$ , respectively, at the end of some finite prefix of some execution, then  $y_i \neq y_j$ .*

The following Lemmas are used to give an upper bound of the number of sieves to which each process visits.

**Lemma 3.** *If one or more processes enter a copy  $c$  of a sieve  $s$  in some round, at least one process obtains a snapshot by **partial\_scan** in  $c$  in this round.*

*Proof.* We prove the lemma by contradiction. Assume that no process obtain a snapshot. In this case, no process writes non-empty set to a variable  $s.list[c].view[sp]$  for any splitter  $sp$ , obtains no-empty set by **candidates**, and writes  $nextDB$  to the variable

$s.allDone[nextC]$ . Since a copy is initialized after  $s.allDone[nextC]$  is set to  $nextDB$ , no process initializes the copy. Let  $p_i$  be the last process which writes its identifier to a variable  $s.list[c].id[sp]$  of some splitter  $sp$  in the copy  $c$  in **register**. The process  $p_i$  then executes **collect** twice in **partial\_scan**. Since a set of processes which have registered does not changes after  $p_i$  registered,  $p_i$  can obtains a snapshot. A contradiction.

**Lemma 4.** *If one or more processes enter a copy  $c$  of a sieve  $s$  in some round, at least one process wins in  $c$  in this round.*

*Proof.* Lemma 3 shows that at least one process obtains a snapshot by **partial\_scan** in  $c$  in this round. Let  $W$  be the minimum snapshot obtained in  $c$  in this round, and  $p_i$  be the last process in  $W$  which writes a value to  $s.list[c].view[sp]$  in some splitter  $sp$ . Since  $W$  is the minimum, every process  $p_j$  in  $W$  obtains a snapshot not smaller than  $W$  or fails to obtain a snapshot,  $p_j$  writes a view  $W'$  in its splitter such that  $W \subseteq W'$  or  $W = \emptyset$ . The process  $p_i$  can see these values in **candidate** and return  $W$  including  $p_i$ . That is,  $p_i$  wins in  $c$  in this round.

The above Lemma 5 is used to show similar Lemmas to Lemmas 3.1 and 3.4 in [11], and we can show the following.

**Lemma 5.** *Every process  $p$  wins in sieve at most  $2k - 1$ , where  $k$  is the point contention of  $p$ 's interval of **getName**.*

The step complexity of the  $O(k^2)$ -algorithm is as follows. In **getName**, each process  $p_i$  visits to at most  $2k - 1$  sieves, and has access to and enters at most one copy in each sieve. For each copy,  $p_i$  invokes one **register**, two **collect**, and at most one **clear**. Each procedure has  $O(k)$  step complexity, and therefore, total step complexity is  $O(k^2)$ . The algorithm uses  $2n - 1$  sieves,  $2N$  copies of each sieve, and  $O(n)$  registers for each copy. Therefore, the space complexity is  $O(n^2 N)$ .

**Theorem 1.** *The  $O(k^2)$ -algorithm solves the point contention adaptive long-lived  $(2k^2 - k)$ -renaming problem with  $O(k^2)$  step complexity and  $O(n^2 N)$  space complexity using bounded values.*

## 6 Conclusions

We presented a long-lived  $(2k^2 - k)$ -renaming algorithm which is adaptive to point contention  $k$  and uses bounded memory and bounded values in the read/write shared memory model. The step complexity is  $O(k^2)$  and the space complexity is  $O(n^2 N)$ , where  $n$  and  $N$  are an upper bound of  $k$  and the number of processes, respectively.

One of the open problems is whether we can develop an adaptive long-lived renaming algorithm where the number of registers used in the algorithm depends on not  $N$  but  $n$ .



## References

1. Y.Afek, H.Attiya, A.Fouren, G.Stupp, D.Touitou: Adaptive long-lived renaming using bounded memory. (1999) Available at [www.cs.technion.ac.il/~hagit/pubs/AAFST99disc.ps.gz](http://www.cs.technion.ac.il/~hagit/pubs/AAFST99disc.ps.gz).
2. Y.Afek, G.Stupp, D.Touitou: Long-lived and adaptive collect with applications. In Proc. 40th IEEE Symp. Foundations of Comp. Sci. (1999) 262–272
3. Y.Afek, G.Stupp, D.Touitou: Long-lived adaptive atomic snapshot and immediate snapshot. In Proc. 19th ACM Symp. Principles of Dist. Comp. (2000) 71–80
4. A.Bar-Noy, D.Dolev: Shared memory versus message-passing in an asynchronous distributed environment. Proc. of the 8th Annual ACM Symposium on Principles of Distributed Computing (1989) 307–318
5. E.Borowsky, E.Gafni: Immediate atomic snapshots and fast renaming. In Proc. 12th ACM Symp. Principles of Dist. Comp. (1993) 41–52
6. M.Moir, J.H.Anderson: Fast, long-lived renaming. In Proc. 8th Int. Workshop on Dist. Algorithms (1994) 141–155
7. H.Buhrman, J.A.Garay, J.H.Hoepman, M.Moir: Long-lived renaming made fast. In Proc. 14th ACM Symp. Principles of Dist. Comp. (1995) 194–203
8. M.Moir, J.A.Garay: Fast long-lived renaming improved and simplified. In Proc. 10th Int. Workshop on Dist. Algorithms (1996) 287–303
9. H.Attiya, A.Fouren: Adaptive wait-free algorithms for lattice agreement and renaming. In Proc. 17th ACM Symp. Principles of Dist. Comp. (1998) 277–286
10. Y.Afek, M.Merritt: Fast, wait-free  $(2k-1)$ -renaming. In Proc. 18th ACM Symp. Principles of Dist. Comp. (1999) 105–112
11. Y.Afek, H.Attiya, A.Fouren, G.Stupp, D.Touitou: Long-lived renaming made adaptive. In Proc. 18th ACM Symp. Principles of Dist. Comp. (1999) 91–103
12. H.Attiya, A.Fouren: Polynomial and adaptive long-lived  $(2k-1)$ -renaming. In Proc. 14th Int. Symp. on Dist. Comp. (2000)
13. H.Attiya, A.Fouren: An adaptive collect algorithm with applications. (1999) Available at [www.cs.tehnion.ac.il/~hagit/pubs/AF99ful.ps.gz](http://www.cs.tehnion.ac.il/~hagit/pubs/AF99ful.ps.gz).
14. Herlihy, M.: Wait-free synchronization. ACM Trans. on Programming Languages and Systems **13** (1991) 124–149

# A New Synchronous Lower Bound for Set Agreement

Maurice Herlihy<sup>1</sup>, Sergio Rajsbaum<sup>2</sup>, and Mark Tuttle<sup>2</sup>

<sup>1</sup> Brown University, Computer Science Department, Providence, RI 02912;  
mph@cs.brown.edu.

<sup>2</sup> Compaq, Cambridge Research Lab, One Cambridge Center, Cambridge, MA 02142;  
Sergio Rajsbaum is on leave from U.N.A.M., Mexico City;  
sergio.rajsbaum@compaq.com, mark.tuttle@compaq.com.

**Abstract.** We have a new proof of the lower bound that  $k$ -set agreement requires  $\lfloor f/k \rfloor + 1$  rounds in a synchronous, message-passing model with  $f$  crash failures. The proof involves constructing the set of reachable states, proving that these states are highly connected, and then appealing to a well-known topological result that high connectivity implies that set agreement is impossible. We construct the set of reachable states in an iterative fashion using a round operator that we define, and our proof of connectivity is an inductive proof based on this iterative construction and using simple properties of the round operator. This is the shortest and simplest proof of this lower bound we have seen.

## 1 Introduction

The consensus problem [20] has received a great deal of attention. In this problem,  $n + 1$  processors begin with input values, and all must agree on one of these values as their output value. Fischer, Lynch, and Paterson [11] surprised the world by showing that solving consensus is impossible in an asynchronous system if one processor is allowed to fail. This leads one to wonder if there is any way to weaken consensus to obtain a problem that can be solved in the presence of  $k - 1$  failures but not in the presence of  $k$  failures. Chaudhuri [6] defined the  $k$ -set agreement problem and conjectured that this was one such problem, and a trio of papers [4, 16, 21] proved that she was right. The  $k$ -set agreement problem is like consensus, but we relax the requirement that processors agree: the set of output values chosen by the processors may contain as many as  $k$  distinct values, and not just 1. Consensus and set agreement are just as interesting in synchronous models as they are in asynchronous models. In synchronous models, it is well-known that consensus requires  $f + 1$  rounds of communication if  $f$  processors can crash [10, 8, 9], and that  $k$ -set agreement requires  $\lfloor f/k \rfloor + 1$  rounds [7]. These lower bounds agree when  $k = 1$  since consensus is just 1-set agreement. In this paper, we give a new proof of the  $\lfloor f/k \rfloor + 1$  lower bound for set agreement in the synchronous message-passing model with crash failures.

All known proofs for the set agreement lower bound depend — either explicitly or implicitly — on a deep connection between computation and topology.

These proofs essentially consider the simplicial complex representing all possible reachable states of a set agreement protocol, and then argue about the connectivity of this complex. These lower bounds for set agreement follow from the observation that set agreement cannot be solved if the complex of reachable states is sufficiently highly-connected. This connection between connectivity and set agreement has been established both in a generic way [14] and in ways specialized to particular models of computation [2,4,7,13,14,15,21]. Once the connection has been established, however, the problem reduces to reasoning about the connectivity of a protocol’s reachable complex.

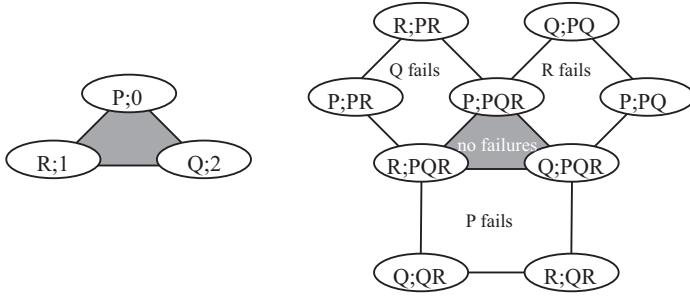
Most of the prior work employing topological arguments has focused on the *asynchronous* model of computation, in which processors can run at arbitrary speeds, and fail undetectably. Reasoning about connectivity in the asynchronous model is simplified by the fact that the connectivity of the reachable complex remains unchanged over time. Moreover, the extreme flexibility of the processor failure model facilitates the use of invariance arguments to prove connectivity. In the *synchronous* model that we consider here, analyzing connectivity is significantly more complicated. The difficulty arises because the connectivity of the reachable complex changes from round to round, so the relatively simple invariance arguments used in the asynchronous model cannot possibly work here.

The primary contribution of this work is a new, substantially simpler proof of how the connectivity of the synchronous complex evolves over time. Our proof depends on two key insights:

1. The notion of a *round operator* that maps a global state to the set of global states reachable from this state by one round of computation, an operator satisfying a few simple algebraic properties.
2. The notion of an *absorbing poset* organizing the set of global states into a partial order, from which the connectivity proof follows easily using the round operator’s algebraic properties.

We believe this new proof has several novel and elegant features. First, we are able to isolate a small set of elementary combinatorial properties of the round operator that suffice to establish the connection with classical topology. Second, these properties require only local reasoning about how the computation evolves from one round to the next. Finally, most connectivity arguments can be difficult to follow because they mix semantic, combinatorial, and topological arguments, but those arguments are cleanly separated here: The definition of the round operator captures the semantics of the synchronous model, the reasoning about the round operator is purely combinatorial, and the lower bound is completed with a “black box” application of well-known topological results without any need to make additional topological arguments.

In the next section, we give an overview of our proof strategy and discuss its relationship to other proofs appearing in the literature. In the main body of the paper, we sketch the proof itself. The full proof in the full paper fills just over a dozen pages, making it the shortest self-contained proof of this lower bound that we have seen.



**Fig. 1.** A global state  $S$  and the set  $\mathcal{R}_1(S)$  of global states after one round from  $S$ .

## 2 Overview

We assume a standard synchronous message-passing model with crash failures [3, 17]. The system has  $n + 1$  processors, and at most  $f$  of them can crash in any given execution. Each processor begins in an initial state consisting of its input value, and computation proceeds in a sequence of rounds. In each round, each processor sends messages to other processors, receives messages sent to it by the other processors in that round, performs some internal computation, and changes state. We assume that processors are following a full-information protocol, which means that each processor sends its entire local state to every processor in every round. This is a standard assumption to make when proving lower bounds. A processor can fail by crashing in the middle of a round, in which case it sends its state only to a subset of the processors in that round. Once a processor crashes, it never sends another message after that.

We represent the local state of a processor with a vertex labeled with that processor's id and its local state. We represent a global state as a set of labeled vertexes, labeled with distinct processors, representing the local state of each processor in that global state. In topology, a *simplex* is a set of vertexes, and a *complex* is a set of simplexes that is closed under containment. Applications of topology to distributed computing often assume that these vertexes are points in space and that the simplex is the convex hull of these points in order to be able to use standard topology results. As you read this paper, you might find it helpful to think of simplexes in this way, but in the purely combinatorial work done in this paper, a simplex is just a set of vertexes.

As an example, consider the simplex and complex illustrated in Figure 1. On the left side, we see a simplex representing an initial global state in which processor  $P$ ,  $Q$ , and  $R$  start with input values 0, 2, and 1. Each vertex is labeled with a processor's id and its local state (which is just its input value in this case). On the right we see a complex representing the set of states that arise after one round of computation from this initial state if one processor is allowed to crash. The labeling of the vertexes is represented schematically by a processor id such as  $P$  and a string of processor ids such as  $PQ$ . The string  $PQ$  is intended to

represent the fact that  $P$  heard from processors  $P$  and  $Q$  during the round but not from  $R$ , since  $R$  failed that round. (We are omitting input values on the right for notational simplicity.) The simplexes that represent states after one round are the 2-dimensional triangle in the center and the 1-dimensional edges that radiate from the triangle (including the edges of the triangle itself). The central triangle represents the state after a round in which no processor fails. Each edge represents a state after one processor failed. For example, the edge with vertexes labeled  $P;PQR$  and  $Q;PQ$  represent the global state after a round in which  $R$  fails by sending a message to  $P$  and not sending to  $Q$ :  $P$  heard from all three processors, but  $Q$  did not hear from  $R$ .

What we do in this paper is define round operators like the round operator  $\mathcal{R}_1$  that maps the simplex  $S$  on the left of Figure 1 to the complex  $\mathcal{R}_1(S)$  on the right, and then argue about the connectivity of  $\mathcal{R}_1(S)$ . Informally, connectivity in dimension 0 is just ordinary graph connectivity, and connectivity in higher dimensions means that there are no “holes” of that dimension in the complex. When we reason about connectivity, we often talk about the connectivity of a simplex  $S$  when we really mean the connectivity of the induced complex consisting of  $S$  and all of its faces. For example, both of the complexes in Figure 1 are 0-connected since they are connected in the graph theoretic sense. In fact, the complex on the left is also 1-connected, but the complex on the right is not since there are “holes” formed by the three cycles of 1-dimensional edges. The fundamental connection between  $k$ -set agreement and connectivity is that  $k$ -set agreement cannot be solved after  $r$  rounds of computation if the complex of states reachable after  $r$  rounds of computation is  $(k - 1)$ -connected. In the remainder of this overview, we sketch how we define a round operator, and how we reason about the connectivity of the complex of reachable states.

## 2.1 Round Operators

In the synchronous model, we can represent a round of computation with a round operator  $\mathcal{R}_\ell$  that maps the state  $S$  at the start of a round to the set  $\mathcal{R}_\ell(S)$  of all possible states at the end of a round in which at most  $\ell$  processors fail. Suppose  $F$  is the set of processors that fail in a round, and consider the local state of a processor  $p$  at the end of that round. The full-information protocol has each processor send its local state to  $p$ , so  $p$  receives the local state of each processor, with the possible exception of some processors in  $F$  that fail before sending to  $p$ . Since each processor  $q$  sending to  $p$  sends its local state, and since this local state labels  $q$ 's vertex in  $S$ , we can view  $p$ 's local state at the end of the round as the face of  $S$  containing the local states  $p$  received from processors like  $q$ . If we define  $S/F$  to be the face of  $S$  obtained by deleting the vertexes of  $S$  labeled with processors in  $F$ , then  $p$  receives at least the local states labeling  $S/F$ , so  $p$ 's local state after the round of computation can be represented by some face of  $S$  containing  $S/F$ .

This intuition leads us to define the round operator  $\mathcal{R}_\ell$  as follows. For each set  $F$  of at most  $\ell$  processors labeling a state  $S$ , define  $\mathcal{R}_F(S)$  to be the set of

simplexes obtained by labeling each vertex of  $S/F$  with some face of  $S$  containing  $S/F$ . This is the set of possible states after a round of computation from  $S$  in which  $F$  is the set of processors that fail, since the processors labeling  $S/F$  are the processors that are still alive at the end of the round, and since they each hear from some set of processors that contains the processors labeling  $S/F$ . The round operator  $\mathcal{R}_\ell(S)$  is defined to be the union of all  $\mathcal{R}_F(S)$  such that  $F$  is a set of at most  $\ell$  processors labeling  $S$ .

To illustrate this informal definition, consider the complex  $\mathcal{R}_1(S)$  of global states on the right of Figure 1. This complex is the union of four rather degenerate pseudospheres, which are complexes defined in Section 5.1 that are topologically similar to a sphere. The first pseudosphere is the central triangle. This is the pseudosphere  $\mathcal{R}_\emptyset(S)$ , where each processor hears from all other processors, so each processor's local state at the end of the round is the complete face  $\{P, Q, R\}$  of  $S$ . The other three pseudospheres are the cycles hanging off the central triangle. These are the pseudospheres of the form  $\mathcal{R}_{\{P\}}(S)$ , where each processor hears from all processors with the possible exception of  $P$ , so each processor's local state at the end of the round is either the whole simplex  $S = \{P, Q, R\}$  or the face  $S/\{P\} = \{Q, R\}$ , depending on whether the processor did or did not hear from  $P$ .

If  $\mathcal{R}_\ell(S)$  is the set of possible states after one round of computation, then  $\mathcal{R}_\ell^r(S) = \mathcal{R}_\ell \mathcal{R}_\ell^{r-1}(S)$  is the set of possible states after  $r$  rounds of computation. The goal of this paper is to prove that  $\mathcal{R}_\ell^r(S)$  is highly-connected.

## 2.2 Absorbing Posets

To illustrate the challenge of proving that  $\mathcal{R}_\ell^r(S)$  is connected, let us assume that  $\mathcal{R}_\ell(S)$  is  $\ell$ -connected for every  $S$  and  $\ell$ , and let us prove that  $\mathcal{R}_\ell \mathcal{R}_\ell(S)$  is  $\ell$ -connected. If  $\mathcal{R}_\ell(S) = \{S_1, \dots, S_k\}$  is the set of states after one round, then

$$\begin{aligned} \mathcal{R}_\ell \mathcal{R}_\ell(S) &= \mathcal{R}_\ell(S_1 \cup S_2 \cup \dots \cup S_k) \\ &= \mathcal{R}_\ell(S_1) \cup \mathcal{R}_\ell(S_2) \cup \dots \cup \mathcal{R}_\ell(S_k) \end{aligned}$$

is the set of states after two rounds. We know that the  $\mathcal{R}_\ell(S_i)$  are  $\ell$ -connected by assumption, but we need to prove that their union is  $\ell$ -connected.

Proving that a union of complexes is connected is made easier by the Mayer-Vietoris theorem, which says that  $A \cup B$  is  $c$ -connected if  $A$  and  $B$  are  $c$ -connected and  $A \cap B$  is  $(c-1)$ -connected. This suggests that we proceed by induction on  $i$  to prove that

$$\mathcal{R}_\ell(S_1) \cup \mathcal{R}_\ell(S_2) \cup \dots \cup \mathcal{R}_\ell(S_i)$$

is connected for  $i = 1, \dots, k$ . We know that

$$\mathcal{R}_\ell(S_1) \cup \mathcal{R}_\ell(S_2) \cup \dots \cup \mathcal{R}_\ell(S_{i-1}) \text{ and } \mathcal{R}_\ell(S_i)$$

are both  $\ell$ -connected by hypothesis and assumption, so all we need to do is prove that their intersection

$$\begin{aligned} &[\mathcal{R}_\ell(S_1) \cup \mathcal{R}_\ell(S_2) \cup \dots \cup \mathcal{R}_\ell(S_{i-1})] \cap \mathcal{R}_\ell(S_i) \\ &= [\mathcal{R}_\ell(S_1) \cap \mathcal{R}_\ell(S_i)] \cup [\mathcal{R}_\ell(S_2) \cap \mathcal{R}_\ell(S_i)] \cup \dots \cup [\mathcal{R}_\ell(S_{i-1}) \cap \mathcal{R}_\ell(S_i)] \end{aligned}$$

is  $(\ell - 1)$ -connected. This union suggests another Mayer-Vietoris argument, but what do we know about the connectivity of the  $\mathcal{R}_\ell(S_j) \cap \mathcal{R}_\ell(S_i)$ ?

One of the elegant properties of the round operator is that

$$\mathcal{R}_\ell(S_j) \cap \mathcal{R}_\ell(S_i) = \mathcal{R}_{\ell-c}(S_j \cap S_i)$$

where  $c$  is the number of vertices in  $S_i$  or  $S_j$  that do not appear in  $S_j \cap S_i$ , whichever number is larger. We refer to this number as the codimension of  $S_i$  and  $S_j$ , and it is a measure of how much the two states have in common. The  $\mathcal{R}_{\ell-c}(S_j \cap S_i)$  are  $(\ell - c)$ -connected by our assumption, but we need to prove that they are  $(\ell - 1)$ -connected for our inductive argument to go through, and it is not generally true that the  $S_i$  and  $S_j$  have codimension  $c = 1$ .

One of the insights in this paper — and one of the reasons that the lower bound proof for set agreement is now so simple — is that we can organize the inductive argument so that we need only consider pairs of simplexes  $S_i$  and  $S_j$  in this union that have codimension  $c = 1$ . If we order the set  $\mathcal{R}_\ell(S) = \{S_1, \dots, S_k\}$  of one-round states correctly, then we can prove that every set  $\mathcal{R}_\ell(S_j) \cap \mathcal{R}_\ell(S_i)$  in the union is contained in another set  $\mathcal{R}_\ell(T_j) \cap \mathcal{R}_\ell(S_i)$  in the union such that  $T_j$  and  $S_i$  have codimension  $c = 1$ . The larger set “absorbs” the smaller set, and while the smaller set may not have the desired  $(\ell - 1)$ -connectivity, the larger set does. Now we can write this union as the union of the absorbing sets, which is a union of  $(\ell - 1)$ -connected sets, and apply Mayer-Vietoris to prove that the union itself is  $(\ell - 1)$ -connected. In this paper, we show how to define a partial order on the set  $\mathcal{R}_\ell(S) = \{S_1, \dots, S_k\}$  of one-round states that guarantees this absorption property holds during the Mayer-Vietoris argument. We call this partial order an *absorbing poset*.

To illustrate the notion of an absorbing poset, consider once again the complex  $\mathcal{R}_1(S)$  of global states on the right of Figure 1. Suppose we order the pseudospheres making up  $\mathcal{R}_1(S)$  by ordering the central triangle first and then ordering the cycles surrounding this triangle in some order. Within each cycle, let us order the edges of the cycle by ordering the edge of the central triangle first, then the two edges intersecting this edge in some order, and finally outermost edge that does not intersect the central triangle. To see that this ordering has the properties of an absorbing poset, consider the central triangle  $T$  and the edge  $E$  consisting of the vertexes  $P; PR$  and  $R; PQR$ . The simplexes  $T$  and  $E$  intersect in the single vertex  $R; PQR$  and hence have codimension two. On the other hand, consider the edge  $F$  consisting of the vertexes  $R; PQR$  and  $P; PQR$ . This edge  $F$  appears between  $T$  and  $E$  in the simplex ordering, the intersection of  $F$  and  $E$  is actually equal to the intersection of  $T$  and  $E$ , and the codimension of  $F$  and  $E$  is one. This property of an absorbing poset is key to the simplicity of the connectivity argument given in this paper.

### 2.3 Related Work

We are aware of three other proofs of the  $k$ -set agreement lower bound.

Chaudhuri, Herlihy, Lynch, and Tuttle [7] gave the first proof. Their proof consisted of taking the standard similarity chain argument used to prove the

consensus synchronous lower bound and running that argument in  $k$  dimensions at once to construct a subset of the reachable complex to which a standard topological tool called Sperner’s Lemma can be applied to obtain the desired impossibility. While their intuition is geometrically compelling, it required quite a bit of technical machinery to nail down the details.

Herlihy, Rajsbaum, and Tuttle [15] gave a proof closer to our “round-by-round” approach. In fact, the round operator that we define here is exactly the round operator they defined. Their connectivity proof for the reachable complex was not easy, however, and the inductive nature of the proof did not reflect the iterative nature of how the reachable complex is constructed by repeatedly applying the round operator locally to a global state  $S$ . The notion of an absorbing poset used in this paper dramatically simplifies the connectivity proof.

Gafni [12] gave another proof in an entirely different style. His proof is based on simple reductions between models, showing that the asynchronous model can simulate the first few rounds of the synchronous model, and thus showing that the synchronous lower bound follows from the known asynchronous impossibility result for set agreement [4,16,21]. While his notion of reduction is elegant, his proof depends on the asynchronous impossibility result, and that result is not easy to prove. We are interested in a simple, self-contained proof that gives as much insight as possible into the topological behavior of the synchronous model of computation.

Round-by-round proofs that show how the 1-dimensional (graph) connectivity evolves in the synchronous model have been described by Aguilera and Toueg [1] and Moses and Rajsbaum [18] (the latter do it in a more general way that applies to various other asynchronous models as well) to prove consensus impossibility results. These show how to do an elegant FLP style of argument, as opposed to the more involved backward inductive argument of the standard proofs [10,8,9]. They present a (graph) connectivity proof of the successors of a global state. Thus, our proofs are similar to this strategy in the particular case of  $k = 1$ , but give additional insights because they show more general ways of organizing these connectivity arguments.

There are also various set agreement impossibility results for asynchronous systems that are related to our work.

Attiya and Rajsbaum [2] and Borowsky and Gafni [4] present two similar proofs for the set agreement impossibility. The relation to our work is that they are also combinatorial. However, their proofs are for an asynchronous, shared memory model. Also, they do not have a round-by-round structure; instead they work by proving that the set of global states at the end of the computation has some properties (somewhat weaker than connectivity) that are sufficient to apply Sperner’s Lemma and obtain the desired impossibility result.

Borowsky and Gafni [5] defined an asynchronous shared-memory model where variables can be used only once, a model they showed to be equivalent to general asynchronous shared-memory models. They defined a round operator as we do, and they showed that one advantage of their model was that it had a very regular iterative structure that greatly simplified computing its connec-



tivity. Unfortunately, their elegant techniques for the asynchronous model do not extend to the synchronous model. Reasoning about connectivity is harder in the synchronous model than the asynchronous model for two reasons. First, the connectivity never decreases in the asynchronous model, whereas it does in the synchronous model, so their techniques cannot extend to our model. Second, processors never actually fail in their construction since dead processors can be modeled as slow processors, but this is not an option in our model, and we are forced to admit simplexes of many dimensions as models of global states where processors have failed.

### 3 Topology

We now give formal definitions of the topological ideas sketched in the introduction.

A *simplex* is just a set of vertexes. Each vertex  $v$  is labeled with a processor id  $id(v)$  and a value  $val(v)$ . We assume that the vertexes of a simplex are labeled with distinct processor ids, and we assume a total ordering  $\leq_{id}$  on processor ids, which induces an ordering on the vertexes of a simplex. A *face* of a simplex is a subset of the simplex's vertexes, and we write  $F \subseteq S$  if  $F$  is a face of  $S$ . A simplex  $X$  is *between* two simplexes  $S_0$  and  $S_1$  if  $S_0 \subseteq X \subseteq S_1$ . A *complex* is a set of simplexes closed under containment (which means that if a simplex belongs to a complex, then so do its faces). If  $\mathcal{A}$  is a set of simplexes, denote by  $\|\mathcal{A}\|$  the smallest simplicial complex containing every simplex of  $\mathcal{A}$ . It is easy to show that

$$\|\mathcal{A} \cup \mathcal{B}\| = \|\mathcal{A}\| \cup \|\mathcal{B}\| \quad \text{and} \quad \|\mathcal{A} \cap \mathcal{B}\| \subseteq \|\mathcal{A}\| \cap \|\mathcal{B}\|.$$

The *codimension* of a set  $\mathcal{S} = \{S_1, \dots, S_m\}$  of simplexes is

$$codim(\mathcal{S}) = \max_i \{\dim(S_i) - \dim(\cap_j S_j)\} = \max_i \{|S_i - \cap_j S_j|\},$$

where  $\dim(\emptyset) = -1$  is the dimension of the empty simplex. This definition satisfies several simple properties, such as:

1. If  $X$  is between two simplexes  $S$  and  $T$ , then

$$codim(S, T) = codim(S, X) + codim(X, T).$$

2. If  $codim(S_0, S_i) \leq 1$  for  $i = 1, \dots, m$ , then

$$codim(S_0, S_1, \dots, S_m) \leq m.$$

3. If  $S_1, \dots, S_m$  is a set of simplexes with largest dimension  $N$  and codimension  $c$ , then their intersection  $S_1 \cap \dots \cap S_m$  is a simplex with dimension  $N - c$ .

The *connectivity* of a complex is a direct generalization of ordinary graph connectivity. A complex is 0-connected if it is connected in the graph-theoretic sense, and while the definition of  $k$ -connectivity is more involved, the precise definition does not matter here since our work depends only on two fundamental properties of connectivity:

**Theorem 1.**

1. If  $S$  is a simplex of dimension  $k$ , then the induced complex  $\|S\|$  is  $(k-1)$ -connected.
2. If the complexes  $\mathcal{L}$  and  $\mathcal{M}$  are  $k$ -connected and  $\mathcal{L} \cap \mathcal{M}$  is  $(k-1)$ -connected, then  $\mathcal{L} \cup \mathcal{M}$  is  $k$ -connected.

By convention, a nonempty complex is  $(-1)$ -connected, and every complex is  $(-k)$ -connected for  $k \geq 2$ . The first property follows from the well-known fact that  $\|S\|$  is  $k$ -connected when  $S$  is a nonempty simplex of dimension  $k$  ( $k \geq 0$ ), but when  $S$  is empty ( $k = -1$ ), the best we can say is that  $\|S\|$  is  $(-2)$ -connected (since every complex is  $(-2)$ -connected). The second property above is a consequence of the well-known Mayer-Vietoris sequence which relates the topology of  $\mathcal{L} \cup \mathcal{M}$  with that of  $\mathcal{L}$ ,  $\mathcal{M}$  and  $\mathcal{L} \cap \mathcal{M}$  (for example, see Theorem 33.1 in [19]).

## 4 Computing Connectivity

Computing the connectivity of  $\mathcal{R}_\ell(\mathcal{A})$  for some complex  $\mathcal{A}$  depends on properties of the round operator  $\mathcal{R}_\ell$  and on how the Mayer-Vietoris argument used in the proof is organized. In this section, we define the notion of an  $f$ -operator and the notion of an absorbing poset that structures the Mayer-Vietoris argument by imposing a partial order on simplexes in the complex  $\mathcal{A}$ , and we prove that an  $f$ -operator applied to this partially-ordered complex  $\mathcal{A}$  is connected.

A *simplicial operator*  $\mathcal{Q}$  is a function with an associated *domain*. It maps every simplex  $S$  in its domain to a set  $\mathcal{Q}(S)$  of simplexes, and it extends to sets of simplexes in its domain in the obvious way with  $\mathcal{Q}(\mathcal{A}) = \cup_{S \in \mathcal{A}} \mathcal{Q}(S)$ . Proving the connectivity of  $\mathcal{Q}(\mathcal{A})$  is simplified if  $\mathcal{Q}$  satisfies the following property:

**Definition 1.** Let  $\mathcal{Q}$  be an operator, and let  $f$  be a function that maps each set  $\mathcal{A}$  of simplexes in the domain of  $\mathcal{Q}$  to an integer  $f(\mathcal{A})$ . We say that  $\mathcal{Q}$  is an  $f$ -operator if for every set  $\mathcal{A}$  of simplexes in the domain of  $\mathcal{Q}$

$$\bigcap_{S \in \mathcal{A}} \|\mathcal{Q}(S)\| \text{ is } (f(\mathcal{A}) - c - 1)\text{-connected}$$

where  $c = \text{codim}(\mathcal{A})$ .

To illustrate this definition, consider a single simplex  $S$  of dimension  $k$ , and remember the fundamental fact of topology that  $\|S\|$  is  $(k-1)$ -connected. Now consider two simplexes  $S$  and  $T$  of dimension  $k$  that differ in exactly one vertex and hence have codimension one. Their intersection  $S \cap T$  has dimension  $k-1$ , so  $\|S \cap T\|$  is  $(k-1-1)$ -connected. In fact, we can show that  $\|S\| \cap \|T\|$  is  $(k-1-1)$ -connected and, in general, that  $\|S\| \cap \|T\|$  is  $(k-c-1)$ -connected if  $c$  is the codimension of  $S$  and  $T$ . In other words, the connectivity of their intersection is reduced by their codimension. In the definition above, if we interpret  $f(\mathcal{A})$  as the maximum connectivity of the complexes  $\|\mathcal{Q}(S)\|$  taken over all simplexes  $S$  in  $\mathcal{A}$ , then this definition says that taking the intersection of the  $\|\mathcal{Q}(S)\|$  reduces

the connectivity by the codimension of the  $S$ . As a simple corollary, if we take the identity operator  $I(S) = \{S\}$  and define  $f(\mathcal{A}) = \max_{S \in \mathcal{A}} \dim(S)$  to be the maximum dimension of any simplex in  $\mathcal{A}$ , then we can prove that the identity operator is an  $f$ -operator.

Proving the connectivity of  $\mathcal{Q}(\mathcal{A})$  is simplified if there is a partial order on the simplexes in  $\mathcal{A}$  that satisfies the following absorption property:

**Definition 2.** *Given a simplicial operator  $\mathcal{Q}$  and a nonempty partially-ordered set  $(\mathcal{S}, \preceq)$  of simplexes in the domain of  $\mathcal{Q}$ , we say that  $(\mathcal{S}, \preceq)$  is an absorbing poset for  $\mathcal{Q}$  if for every two simplexes  $S$  and  $T$  in  $\mathcal{S}$  with  $T \not\preceq S$  there is  $T_S \in \mathcal{S}$  with  $T_S \preceq T$  such that*

$$\|\mathcal{Q}(S)\| \cap \|\mathcal{Q}(T)\| \subseteq \|\mathcal{Q}(T_S)\| \cap \|\mathcal{Q}(T)\| \quad (1)$$

$$\text{codim}(T_S, T) = 1. \quad (2)$$

For example, if  $\mathcal{S}$  is totally ordered, then every intersection  $\|\mathcal{Q}(S)\| \cap \|\mathcal{Q}(T)\|$  involving a simplex  $S$  preceding a simplex  $T$  is contained in another intersection  $\|\mathcal{Q}(T_S)\| \cap \|\mathcal{Q}(T)\|$  involving another simplex  $T_S$  preceding  $T$  with the additional property that  $T_S$  and  $T$  have codimension 1.

To see why such an ordering is useful, consider the round operator  $\mathcal{R}_\ell$ , and remember the problem we faced in the overview of proving that

$$\mathcal{R}_\ell(S_1) \cup \mathcal{R}_\ell(S_2) \cup \dots \cup \mathcal{R}_\ell(S_i)$$

is connected for  $i = 1, \dots, k$ . We were concerned that  $\mathcal{R}_\ell(S_j) \cap \mathcal{R}_\ell(S_i) = \mathcal{R}_{\ell-c}(S_j \cap S_i)$  was  $(\ell-c)$ -connected and in general might not be  $(\ell-1)$ -connected since the codimension  $c$  of  $S_j$  and  $S_i$  might be too high. If we can impose an ordering on the  $S_1, \dots, S_k$  and prove that the  $S_1, \dots, S_k$  form an absorbing poset for  $\mathcal{R}_\ell$ , then each  $\mathcal{R}_\ell(S_j) \cap \mathcal{R}_\ell(S_i)$  with  $j < i$  is contained in another  $\mathcal{R}_\ell(S_{j'}) \cap \mathcal{R}_\ell(S_i)$  with  $j' < i$  where  $S_{j'}$  and  $S_i$  have codimension one. This means that when computing the connectivity of the union we can restrict our attention to the intersections  $\mathcal{R}_\ell(S_{j'}) \cap \mathcal{R}_\ell(S_i)$  with codimension one, and the proof goes through.

In general, we can prove that applying an operator to an absorbing poset yields a connected complex:

**Theorem 2.** *If  $\mathcal{Q}$  is an  $f$ -operator and  $(\mathcal{A}, \preceq)$  is an absorbing poset for  $\mathcal{Q}$ , then*

$$\|\mathcal{Q}(\mathcal{A})\| = \bigcup_{S \in \mathcal{A}} \|\mathcal{Q}(S)\| \text{ is } (f-1)\text{-connected}$$

where  $f = \min_{\mathcal{B} \subseteq \mathcal{A}} f(\mathcal{B})$ .

In the special case of the identity operator, we say that  $(\mathcal{A}, \preceq)$  is an absorbing poset if it is an absorbing poset for the identity operator. It is an easy corollary to show that if  $(\mathcal{A}, \preceq)$  is an absorbing poset, then

$$\|\mathcal{A}\| = \bigcup_{S \in \mathcal{A}} \|S\| \text{ is } (N-1)\text{-connected,}$$

where  $N$  is the minimum dimension of the simplexes in  $\mathcal{A}$ .

## 5 Synchronous Connectivity

In this section, we show how to use the ideas of the previous section to prove that  $\mathcal{R}_k^r(S)$  is  $(k-1)$ -connected, from which we conclude that  $k$ -set agreement is impossible to solve in  $r$  rounds.

### 5.1 Round Operators

Given a simplex  $S$  representing the state at the beginning of a round, and given a set  $X$  of processors that fail during the round, let  $F = S/X$  be the face of  $S$  obtained from  $S$  by deleting the vertexes labeled with processors in  $X$ . The set of all possible states at the end of a round of computation from  $S$  in which processors in  $X$  fail can be represented by the set of all possible simplexes obtained by labeling the vertexes of  $F$  with simplexes between  $F$  and  $S$ . This set of simplexes obtained in this way forms a set that we call a pseudosphere. For every simplex  $S$ , the *pseudosphere operator*  $\mathcal{P}_S(F)$  maps a face  $F$  of  $S$  to the set of all labelings of  $F$  with simplexes between  $F$  and  $S$ . The set  $\mathcal{P}_S(F)$  is called a *pseudosphere*, and the face  $F$  is called the *base simplex* of the pseudosphere. Given a simplex  $T$  contained in a pseudosphere  $\mathcal{P}_S(F)$  we define  $\text{base}(T)$  to be the base simplex  $F$  of the pseudosphere.

If  $\ell$  processors fail during the round, there are many ways to choose this set  $X$  of processors that fail, and hence many ways to choose the base simplexes  $F = S/X$  for the pseudospheres whose simplexes represent the states at the end of the round. For every integer  $\ell \geq 0$ , the  $\ell$ -*failure operator*  $\mathcal{F}_\ell(S)$  maps a simplex  $S$  to the set of all faces  $F$  of  $S$  with  $\text{codim}(F, S) \leq \ell$ , which is the set of all faces obtained by deleting at most  $\ell$  vertexes from  $S$ . The domain of the operator  $\mathcal{F}_\ell(S)$  is the set of all simplexes  $S$  with  $\dim(S) \geq \ell$ .

Finally, for every integer  $\ell \geq 0$ , the *synchronous round operator*  $\mathcal{R}_\ell(S)$  is defined by

$$\mathcal{R}_\ell(S) = \mathcal{P}_S(\mathcal{F}_\ell(S)).$$

The domain of this operator  $\mathcal{R}_\ell(S)$  is the set of all simplexes  $S$  with  $\dim(S) \geq \ell + k$ . This round operator satisfies a number of basic properties such as:

**Lemma 1.**

1.  $\mathcal{R}_\ell(S) \subseteq \mathcal{R}_m(S)$  if  $\ell \leq m$  and  $S$  is in the domain of  $\mathcal{R}_m$ .
2.  $\mathcal{R}_\ell(S) \subseteq \mathcal{R}_{\ell+c}(T)$  if  $S \subseteq T$  and  $c = \text{codim}(S, T)$ .
3.  $\mathcal{R}_\ell(S_1) \cap \cdots \cap \mathcal{R}_\ell(S_m) = \mathcal{R}_{\ell-c}(S_1 \cap \cdots \cap S_m)$  if  $c = \text{codim}(S_1, \dots, S_m)$ ,  $\ell \geq c$ , and each  $S_i$  is in the domain of  $\mathcal{R}_\ell$ .
4.  $\|\mathcal{R}_\ell(S_1)\| \cap \cdots \cap \|\mathcal{R}_\ell(S_m)\| = \|\mathcal{R}_\ell(S_1) \cap \cdots \cap \mathcal{R}_\ell(S_m)\|$ .

*Proof.* We sketch the proof of property 3. □

For the  $\supseteq$  containment, suppose  $A \in \mathcal{R}_{\ell-c}(\cap_j S_j)$ . This means that  $A$  is a labeling of a simplex  $F$  with simplexes between  $F$  and  $\cap_j S_j$  for some face  $F$  of  $\cap_j S_j$  satisfying  $\text{codim}(F, \cap_j S_j) \leq \ell - c$ . Since  $A$  is a labeling of  $F$  with simplexes between  $F$  and  $\cap_j S_j$ , it is obviously a labeling of  $F$  with simplexes

between  $F$  and  $S_i$ . We have  $A \in \mathcal{R}_\ell(S_i)$  since  $F$  is a face of  $\cap_j S_j$  which is in turn a face of  $S_i$ , and hence

$$\begin{aligned} \text{codim}(F, S_i) &= \text{codim}(F, \cap_j S_j) + \text{codim}(\cap_j S_j, S_i) \\ &\leq \text{codim}(F, \cap_j S_j) + \text{codim}(S_1, \dots, S_m) \\ &\leq (\ell - c) + c = \ell. \end{aligned}$$

For the  $\subseteq$  containment, suppose  $A \in \cap_j \mathcal{R}_\ell(S_j)$ . For each  $i$ , we know that  $A$  is a labeling of  $F_i$  with simplexes between  $F_i$  and  $S_i$  for some face  $F_i$  of  $S_i$  satisfying  $\text{codim}(F_i, S_i) \leq \ell$ . Since  $A$  is a labeling of  $F_i$  for each  $i$ , it must be that the  $F_i$  are all equal, so let  $F$  be this common face of the  $S_i$  and hence of  $\cap_j S_j$ . Since  $A$  is a labeling of  $F$  with simplexes between  $F$  and  $S_i$  for each  $i$ , it must be that  $A$  is a labeling of  $F$  with simplexes between  $F$  and  $\cap_j S_j$ . Since  $F$  is a face of  $\cap_j S_j$  which is in turn a face of each  $S_i$ , including any  $S_M$  satisfying  $\text{codim}(\cap_j S_j, S_M) = \text{codim}(S_1, \dots, S_M) = c$ , we have  $A \in \mathcal{R}_{\ell-c}(\cap_j S_j)$  since  $\text{codim}(F, \cap_j S_j) = \text{codim}(F, S_M) - \text{codim}(\cap_j S_j, S_M) \leq \ell - c$ .  $\square$

The round operator  $\mathcal{R}_\ell$  models a single round of computation. We model multiple rounds of computation with the multi-round operator  $\mathcal{R}_L^r \mathcal{R}_\ell$  defined inductively by  $\mathcal{R}_L^0 \mathcal{R}_\ell(S) = \mathcal{R}_\ell(S)$  and  $\mathcal{R}_L^r \mathcal{R}_\ell(S) = \mathcal{R}_L(\mathcal{R}_L^{r-1} \mathcal{R}_\ell(S))$  for  $r > 0$ . The domain of  $\mathcal{R}_L^r \mathcal{R}_\ell$  is the set of all simplexes  $S$  with  $\dim(S) \geq rL + \ell + k$ . The properties of one-round operators given above generalize to multi-round operators where  $\mathcal{R}_\ell$  is replaced by  $\mathcal{R}_L^r \mathcal{R}_\ell$ .

## 5.2 Absorbing Posets

We now impose a partial order on  $\mathcal{R}_\ell(S)$  and prove that it is an absorbing poset. First we order the pseudospheres  $\mathcal{P}_S(F)$  making up  $\mathcal{R}_\ell(S)$ , and then we order the simplexes within each pseudosphere  $\mathcal{P}_S(F)$ .

Both of these orders depend on ordering the faces  $F$  of  $S$ , which we do lexicographically. First we order the faces  $F$  by decreasing dimension, so that large faces occur before small faces. Then we order faces of the same dimension with a rather arbitrary rule using on our total order on processor ids: we order  $F_0$  before  $F_1$  if the smallest processor id labeling vertexes in  $F_0$  and not  $F_1$  comes before the smallest processor id labeling  $F_1$  and not  $F_0$ . Formally:

**Definition 3.** Define the total order  $\leq_f$  on the faces of a simplex  $S$  by  $F_0 \leq_f F_1$  iff

1.  $\dim(F_0) > \dim(F_1)$  or
2.  $\dim(F_0) = \dim(F_1)$  and either
  - a)  $F_0 = F_1$  or
  - b)  $F_0 \neq F_1$  and  $p_0 <_{id} p_1$ 

where  $p_0 = \min \{ids(F_0) - ids(F_1)\}$  and  $p_1 = \min \{ids(F_1) - ids(F_0)\}$ .

This face ordering induces an ordering on pseudospheres:  $\mathcal{P}_S(F_0)$  comes before  $\mathcal{P}_S(F_1)$  if  $F_0$  comes before  $F_1$  in the face ordering. This face ordering also

induces an ordering on the simplexes within a single pseudosphere  $\mathcal{P}_S(F)$ :  $S_0$  comes before  $S_1$  if for each vertex  $v$  of the base simplex  $F$  the face labeling  $v$  in  $S_0$  comes before the face labeling  $v$  in  $S_1$ . This ordering of  $\mathcal{P}_S(F)$  is defined formally as follows:

**Definition 4.** Define the partial order  $\preceq_p$  on  $\mathcal{P}_S(F)$  by  $S_0 \preceq_p S_1$  iff  $S_{0,v} \preceq_f S_{1,v}$  for each vertex  $v$  in  $F$  where  $S_{0,v}$  and  $S_{1,v}$  are the simplexes labeling the vertex  $v$  in  $S_0$  and  $S_1$ .

Pulling everything together, the partial order on  $\mathcal{R}_\ell(S)$  is defined as follows:

**Definition 5.** Define the partial order  $\preceq_r$  on  $\mathcal{R}_\ell(S)$  by  $S_0 \preceq_r S_1$  iff

1. different pseudospheres:  $\text{base}(S_0) <_f \text{base}(S_1)$  or
2. same pseudosphere:  $\text{base}(S_0) = \text{base}(S_1)$  and  $S_0 \preceq_p S_1$

Now we can prove that  $(\mathcal{R}_\ell(S), \preceq_r)$  is an absorbing poset, and that

**Lemma 2.**  $(\mathcal{R}_\ell(S), \preceq_r)$  is an absorbing poset for  $\mathcal{R}_L^r$  for  $\dim(S) \geq rL + \ell + k$ .

*Proof.* We prove only the base case that  $(\mathcal{R}_\ell(S), \preceq_r)$  is an absorbing poset. Let  $A$  and  $B$  be two simplexes in  $\mathcal{R}_\ell(S)$  satisfying  $B \not\preceq_r A$ .

*Case 1:* Suppose  $A$  and  $B$  are in the same pseudosphere  $\mathcal{P}_S(F)$  for some face  $F$  of  $S$ . The simplexes  $A$  and  $B$  are labelings of  $F$  with simplexes between  $F$  and  $S$ , so let  $A_v$  and  $B_v$  denote the label of vertex  $v$  in  $A$  and  $B$  for every vertex  $v \in F$ . There must be some vertex  $v$  with  $A_v <_f B_v$  since  $B \not\preceq_r A$ . Let  $B_A$  be  $B$  with the label of  $v$  changed from  $B_v$  to  $A_v$ . We have  $B_A \prec_r B$  since the label of  $v$  in  $B_A$  is ordered before the label of  $v$  in  $B$ , and the labels of all other vertices are equal. We have  $A \cap B \subseteq B_A$  since  $v$  is not in  $A \cap B$  due to the conflicting labels for  $v$ , while all other vertexes of  $B$  and hence of  $A \cap B$  are in  $B_A$ . Finally, we have  $\text{codim}(B_A, B) = 1$  since  $B_A$  and  $B$  differ only in the label of  $v$ .

*Case 2:* Suppose  $A$  and  $B$  are in different pseudospheres  $\mathcal{P}_S(F_A)$  and  $\mathcal{P}_S(F_B)$  for distinct faces  $F_A$  and  $F_B$  of  $S$ . We can assume without loss of generality that every vertex of  $B - A$  is labeled with  $S$ , and we can show that  $F_A <_f F_B$ .

*Case 2a:* Suppose  $\dim(F_A) > \dim(F_B)$ . Since  $\dim(F_A) > \dim(F_B)$ , the set  $F_A - F_B$  must be nonempty, so choose any vertex  $v \in F_A - F_B$ . Since  $A \in \mathcal{P}_S(F_A)$ , the simplex  $A$  must be a labeling of  $F_A$  with simplexes between  $F_A$  and  $S$ . Since  $v$  is a vertex of  $F_A$ , this means that  $v$  appears in all simplexes labeling  $A$ , and hence in all simplexes labeling  $A \cap B$ . Since we have assumed that  $S$  is the label of every vertex in  $B - A$ , and since  $S$  certainly contains the vertex  $v$ , the vertex  $v$  appears in all labels of  $B - A$ . It follows that  $v$  appears in every simplex labeling  $B$ . Let  $B_A$  be the simplex consisting of  $B$  together with the vertex  $v$  labeled with  $S$ , and notice that  $B_A$  is a simplex in  $\mathcal{R}_\ell(S)$ . We have  $B_A \prec_r B$  since  $\dim(B_A) = \dim(B) + 1 > \dim(B)$ . We have  $A \cap B \subseteq B_A$  since  $A \cap B \subseteq B \subseteq B_A$ . We have  $\text{codim}(B, B_A) = 1$  since  $B$  and  $B_A$  differ only in  $v$ .

*Case 2b:* Suppose  $\dim(F_A) = \dim(F_B)$ , in which case we have  $p_A \prec_p p_B$  where  $p_A = \min \{\text{ids}(F_A) - \text{ids}(F_B)\}$  and  $p_B = \min \{\text{ids}(F_B) - \text{ids}(F_A)\}$ . Let  $v_A$  and  $v_B$  be the vertexes for processors  $p_A$  and  $p_B$  in the faces  $F_A$  and  $F_B$  of  $S$ . Let  $F_C$  be the face of  $S$  obtained from  $F_B$  by replacing  $v_B$  with  $v_A$ , and

let  $C$  be the labeling of  $F_C$  obtained by labeling  $v_A$  with  $S$  and every other vertex with its label in  $B$ . Since  $A$  is a labeling of  $F_A$  with simplexes between  $F_A$  and  $S$ , and since  $v_A$  is a vertex of  $F_A$ , the vertex  $v_A$  appears in every simplex labeling  $A$  and hence  $A \cap B$ ; and since we are assuming that every vertex of  $B - A$  is labeled with  $S$  which certainly contains  $v_A$ , it follows that every vertex of  $B - A$  is labeled with a simplex containing  $v_A$ ; and hence it follows that every label in  $B$  contains  $F_C$ . It follows that  $C \in \mathcal{R}_\ell(S)$  since  $C$  is a labeling of a face  $F_C$  of  $S$  with simplexes between  $F_C$  and  $S$ . We have  $C \prec_r B$  since

$$\min \{ids(F_C) - ids(F_B)\} = p_A \prec_p p_B = \min \{ids(F_B) - ids(F_C)\}.$$

We have  $A \cap B \subseteq C$  and  $\text{codim}(B, C) = 1$ . Taking  $B_A = C$ , we are done.  $\square$

### 5.3 Connectivity

All that remains is to prove that  $\mathcal{R}_k^r$  is a  $k$ -operator. This follows from the following pair of statements proven by mutual induction:

**Theorem 3.** *For all  $r \geq 0$ ,*

1.  $\|\mathcal{R}_L^r \mathcal{R}_\ell(S)\|$  is  $(k - 1)$ -connected for all  $\ell \geq 0$ , all  $L \geq k$ , and all  $S$  in the domain of  $\mathcal{R}_L^r \mathcal{R}_\ell$ .
2.  $\mathcal{R}_L^r \mathcal{R}_\ell$  is a  $k$ -operator for all  $L, \ell \geq k$ .

Since  $\mathcal{R}_k^r$  is a  $k$ -operator, and since  $(\mathcal{R}_k(S), \preceq_r)$  is an absorbing poset for  $\mathcal{R}_k^r$ , the connectivity follows by Theorem 2.

**Corollary 1.**  $\|\mathcal{R}_k^r(S)\|$  is  $(k - 1)$ -connected if  $\dim(S) \geq (r + 1)k$ .

## 6 Conclusion

As we have said, the impossibility of  $k$ -set agreement now follows directly from the connectivity of  $\|\mathcal{R}_k^r(S)\|$  using standard arguments based on variants of Sperner's Lemma that have appeared in several places now. We hope that the notions of a round operator and an absorbing poset will yield simple proofs of other results, and will show the way toward simple proofs in other models of computation such as the asynchronous message-passing model.

## References

1. Marcos Kawazoe Aguilera and Sam Toueg. A simple bivalency proof that  $t$ -resilient consensus requires  $t + 1$  rounds. *Information Processing Letters*, 71(3-4):155–158, August 1999.
2. Hagit Attiya and Sergio Rajsbaum. The combinatorial structure of wait-free solvable tasks. In *Proceedings of the 10th International Workshop on Distributed Algorithms*, volume 1151 of *Lecture Notes in Computer Science*, pages 322–343. Springer-Verlag, Berlin, October 1996.

3. Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations and Advanced Topics*. McGraw-Hill, 1998.
4. Elizabeth Borowsky and Eli Gafni. Generalized FLP impossibility result for  $t$ -resilient asynchronous computations. In *Proceedings of the 25th ACM Symposium on Theory of Computing*, pages 91–100, May 1993.
5. Elizabeth Borowsky and Eli Gafni. A simple algorithmically reasoned characterization of wait-free computations. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, pages 189–198, August 1997.
6. Soma Chaudhuri. Agreement is harder than consensus: set consensus problems in totally asynchronous systems. In *Proceedings of the 9th Annual ACM Symposium on Principles of Distributed Computing*, pages 311–234, August 1990.
7. Soma Chaudhuri, Maurice Herlihy, Nancy Lynch, and Mark R. Tuttle. Tight bounds for  $k$ -set agreement. *Journal of the ACM*, 47(5):912–943, September 2000.
8. Danny Dolev and Ray Strong. Polynomial algorithms for multiple processor agreement. In *Proceedings of the 14th ACM Symposium on Theory of Computing*, pages 401–407, 1982.
9. Cynthia Dwork and Yoram Moses. Knowledge and common knowledge in a Byzantine environment: Crash failures. *Information and Computation*, 88(2):156–186, October 1990.
10. Michael J. Fischer and Nancy A. Lynch. A lower bound for the time to assure interactive consistency. *Information Processing Letters*, 14(4):183–186, June 1982.
11. Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. Impossibility of distributed consensus with one faulty processor. *Journal of the ACM*, 32(2):374–382, April 1985.
12. Eli Gafni. A round-by-round failure detector — unifying synchrony and asynchrony. In *Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing*. ACM, June 1998.
13. Maurice Herlihy and Sergio Rajsbaum. Set consensus using arbitrary objects. In *Proceedings of the 13th Annual ACM Symposium on Principles of Distributed Computing*, pages 324–333, August 1994.
14. Maurice Herlihy and Sergio Rajsbaum. Algebraic spans. *Mathematical Structures in Computer Science*, 10(4):549–573, August 2000. Special Issue: Geometry and Concurrency.
15. Maurice Herlihy, Sergio Rajsbaum, and Mark R. Tuttle. Unifying synchronous and asynchronous message-passing models. In *Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing*, pages 133–142. ACM, June 1998.
16. Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *Journal of the ACM*, 46(6):858–923, November 1999.
17. Nancy Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.
18. Yoram Moses and Sergio Rajsbaum. The unified structure of consensus: a layered analysis approach. In *Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing*, pages 123–132. ACM, June 1998. To appear in *SIAM Journal of Computing*.
19. James R. Munkres. *Elements Of Algebraic Topology*. Addison Wesley, Reading MA, 1984.
20. Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980.
21. Michael Saks and Fotios Zaharoglou. Wait-free  $k$ -set agreement is impossible: The topology of public knowledge. *SIAM Journal on Computing*, 29(5):1449–1483, 2000.



# The Complexity of Synchronous Iterative Do-All with Crashes<sup>\*</sup>

Chryssis Georgiou<sup>1</sup>, Alexander Russell<sup>1</sup>, and Alex A. Shvartsman<sup>1,2</sup>

<sup>1</sup> Department of Computer Science and Engineering  
University of Connecticut, Storrs, CT 06269, USA.

<sup>2</sup> Laboratory of Computer Science  
Massachusetts Institute of Technology, Cambridge, MA 02139, USA.

**Abstract.** *Do-All* is the problem of performing  $N$  tasks in a distributed system of  $P$  failure-prone processors [8]. Many distributed and parallel algorithms have been developed for this problem and several algorithm simulations have been developed by iterating *Do-All* algorithms. The efficiency of the solutions for *Do-All* is measured in terms of *work* complexity where all processing steps taken by the processors are counted. We present the first non-trivial lower bounds for *Do-All* that capture the dependence of work on  $N$ ,  $P$  and  $f$ , the number of processor crashes. For the model of computation where processors are able to make perfect load-balancing decisions locally, we also present matching upper bounds. We define the *r-iterative Do-All* problem that abstracts the repeated use of *Do-All* such as found in algorithm simulations. Our  $f$ -sensitive analysis enables us to derive a tight bound for *r-iterative Do-All* work (that is stronger than the  $r$ -fold work complexity of a single *Do-All*). Our approach that models perfect load-balancing allows for the analysis of specific algorithms to be divided into two parts: (i) the analysis of the cost of tolerating failures while performing work, and (ii) the analysis of the cost of implementing load-balancing. We demonstrate the utility and generality of this approach by improving the analysis of two known efficient algorithms. Finally we present a new upper bound on simulations of synchronous shared-memory algorithms on crash-prone processors.

## 1 Introduction

Performing a set of tasks in a decentralized setting is a fundamental problem in distributed computing. This is often challenging because the set of processors available to the computation and their ability to communicate may dynamically change due to perturbations in the computation medium. An abstract statement of this problem, referred to as the *Do-All* problem —  *$P$  fault-prone processors perform  $N$  independent tasks* — is one of the standard problems in the research on the complexity of fault-tolerant distributed computation [8, 17]. This

---

<sup>\*</sup> This research is supported by the NSF Grant 9988304. The work of the second author is supported in part by the NSF CAREER Award 0093065. The work of the third author is supported in part by the NSF CAREER Award 9984774.

problem has been studied in shared-memory models (*Write-All*) [18,19,25], in message-passing models [6,8,10], and in partitionable networks (*Omni-Do*) [7,11,24]. Solutions for *Do-All* must perform all tasks efficiently in the presence of specific failure patterns. The efficiency is assessed in terms of work, time and communication complexity depending on the specific model of computation.

In the design of practical distributed/parallel programs one needs to ensure good performance and dependability under unpredictable load patterns caused by deviations from synchrony or by the failures of some processors to complete tasks on time. Here again, a common challenge is to perform  $N$  independent tasks on  $P$  processors [16]. Such tasks could be copying a large array, searching a collection of data, or applying a function to all elements of a matrix [14].

In this paper we focus on the work complexity of the *Do-All* problem in the presence of arbitrary failure patterns imposed by an adversary. The processors are synchronous and are assumed to be fail-stop [30]. The work complexity reflects the *total* amount of processing steps expended by an algorithm [18]. A distinguishing feature of our results is that the complexity is expressed in terms of the *number of processor crashes*  $f$  in addition to  $P$  and  $N$ .

Our approach is motivated in part by the analyses of the consensus problems. The venerable FLP impossibility result [9] and the algorithms that solve consensus in the models that allow fault-tolerant solutions teach the following: (i) asynchronous models are too weak for fault-tolerance [22], and (ii) the maximum number of processor failures needs to be included in upper/lower bounds and impossibility results, e.g., to tolerate  $f$  failures in some models, consensus algorithms require  $P = 3f + 1$  total processors [23,28]. In this work we consider crash failures to ensure that solutions exist for as long as the number of failures  $f$  is inferior to the number of processors  $P$ , and we aim to express the work of the synchronous processors as a function of  $N$ ,  $P$  and  $f$ .

Until very recently, an unsatisfactory landscape existed with respect to the understanding of how the bounds on work depend on  $f$ , the number of failures. That is, work was typically given as a function of  $N$  and  $P$ , but it was either not elucidated how  $f$  impacts work, or, when  $f$  was a part of the equation, it was primarily due to the nature of a specific algorithm, and not due to the inherent properties of the *Do-All* problem. For example, the work of the best known synchronous shared-memory algorithm [17] is given as a function of  $N$  and  $P$ . This is also the case with the best known asynchronous shared-memory algorithm [2]. Similarly, the best known shared-memory lower bound on work for *Do-All* is not parameterized in terms of  $f$  [19]. Likewise, the best known lower bound applicable to message-passing models does not involve  $f$  [3]. The work of message-passing algorithms, e.g. [6,10], typically includes  $f$ , but this is due to the use of single coordinators, which means that for  $f$  coordinator failures the work necessarily includes a factor  $f \cdot P$ . A message-passing algorithm using multiple coordinators [4] avoids this inefficiency and includes a factor that depends on  $\log f$  (but as we show in this paper, that analysis involves  $f$  in a somewhat superficial way). Thus prior lower/upper bound results for *Do-All* do not teach adequately how the work complexity depends on the number of failures  $f$ .

When considering the synchronous shared-memory computing with failure-prone processors the impact of imprecise analysis of work complexity is especially significant. Approaches such as [21,29] use *iterative Do-All* approach to execute synchronous parallel (PRAM) algorithms on failure-prone processors by simulating the parallel steps of ideal processors with the help of some chosen *Do-All* algorithm (see also related work below). It was shown that the execution of a single  $N$ -processor step on  $P$  failure-prone processors does not exceed the complexity of solving a  $N$ -size instance of *Do-All* using  $P$  failure-prone processors. Thus if  $W_{N,P}$  is the complexity of solving a *Do-All* instance of size  $N$  using  $P$  processors, and the *parallel-time*  $\times$  *processor* product of the given  $N$ -processor algorithm is  $\tau \cdot N$ , then the algorithm can be deterministically simulated with work  $O(\tau \cdot W_{N,P})$ . If the analysis does not accurately reflect the impact of the number of failures  $f$ , then the resulting upper bound is needlessly inflated.

**Contributions.** In this work we study the work complexity of deterministic *Do-All* in the presence of arbitrary dynamic patterns of stop-failures. Let  $N$  be the size of the *Do-All* problem,  $P$  the number of processors, and  $f$  the number of crashes ( $0 \leq f < P \leq N$ ). We present the first *complete* analysis of *Do-All* work complexity under the perfect load balancing assumption by proving matching upper and lower bounds as functions of  $N$ ,  $P$  and  $f$ . This is for the model of computation where the computation is fully abstracted away from the low-level shared-memory and message-passing issues, and where a worst-case omniscient dynamic adversary can cause up to  $f$  crashes. This also establishes the first non-trivial lower bound for *Do-All* for moderate number of failures ( $f \leq P/\log P$ ). An important contribution of this work is the definition and analysis of the *r-iterative Do-All* problem that models the repetitive use of *Do-All* algorithms (such as found in algorithm simulations).

We demonstrate the utility and generality of our results by showing new bounds on work for fault-tolerant simulations of arbitrary PRAM algorithms on crash-prone processors, and by improving the analyses of two known algorithms. We derive a new and complete failure sensitivity analysis of the best known algorithm for the synchronous shared-memory model (algorithm W [17]). We also give an improved analysis of the work and message complexity for an efficient message-passing algorithm (algorithm AN [4]).

We give a detailed summary of the complexity results in Section 2.

**Related work—algorithm simulations.** *Do-All* algorithms can be used iteratively to simulate parallel algorithms formulated for synchronous failure-free processors in deterministic and probabilistic settings [21,19,26,27,29]. This commonly requires that (i) the individual processor steps are made idempotent (since they may have to be performed multiple times due to failures or asynchrony), and that (ii) a linear in the number of processors auxiliary memory is made available (to be used as a “scratchpad” and to store intermediate results). While the former can be solved with the help of an automated tool, e.g., a compiler, the latter requires sophisticated solutions because of the difficulty of (re)using the auxiliary memory due to “late writers” (i.e., processors that are slow and that unknowingly write stale values to memory). Examples of randomized so-

lutions addressing these problems include [11,20]. Another important aspect of algorithm simulations is the use of an optimistic approach, where the computation may proceed for several steps assuming that all tasks assigned to active processors are successfully completed, e.g., [19]. In some deterministic models optimal simulations are possible (cf. [29]), however randomized solutions are able to achieve optimality (whp) for broader ranges of models and algorithms. An example of a practical implementation is discussed in [5].

The rest of the paper is structured as follows. In Section 2 we summarize the results. Section 3 we give models and definitions. In Section 4 we present the bounds under the perfect load-balancing assumption. We give new upper bounds for the message-passing model in Section 5, and for the shared-memory model in Section 6. We conclude in Section 7.

*Due to the page limit, most proofs are either omitted or given as sketches. A full version of this paper is available [13].*

## 2 Grand Tour of the Results

We let  $Do-All(N, P, f)$  stand for the  $Do-All$  problem for  $N$  tasks,  $P$  processors and up to  $f$  failures. We let  $Do-All^{\mathcal{O}}(N, P, f)$  denote the  $Do-All(N, P, f)$  problem that is solved with the use of an omniscient oracle that assists the processors (but unlike the oracle's delphian colleague, it cannot predict the future). The oracle assumption is used as a *tool* for studying the work complexity patterns of *any* fault-tolerant algorithm that implements perfect work-load balancing. This allows for the complexity analysis of specific algorithms to be divided into two parts: (i) the analysis of the cost of tolerating failures while performing work assuming perfect load-balancing, and (ii) the analysis of the cost of implementing perfect load-balancing. We use exactly this approach to derive new  $f$ -sensitive upper bounds for message-passing and shared-memory models.

We have shown [12,18] that  $Do-All^{\mathcal{O}}(N, P, f)$  can be solved with work  $O(N + P \frac{\log P}{\log \log P})$  where  $f < P$ , and gave a matching lower bound in the specific case where  $f = \frac{P}{\log \log P} + O(\frac{P}{(\log \log P)^2})$ . This meant that as long as the adversary can cause at least  $\frac{P}{\log \log P}$  failures,  $Do-All^{\mathcal{O}}(N, P, f)$  has matching upper and lower bounds of  $N + P \frac{\log P}{\log \log P}$ . We also showed that when  $f = o(\frac{P}{\log P})$  then  $Do-All^{\mathcal{O}}(N, P, f)$  can be solved with work  $O(N + P \log_{\max\{2, \frac{P}{2f}\}} P)$ .

Thus prior to our newest results: (i) no non-trivial lower bounds were known for  $f < \frac{P}{\log P}$ , (ii) no  $f$ -sensitive analysis was available for the upper bounds when  $f$  is between  $\frac{P}{\log P}$  and  $\frac{P}{\log \log P}$ , and therefore, (iii) there existed a gap in upper/lower bounds analysis for the range  $1 < f < \frac{P}{\log \log P}$ , where  $f = \omega(1)$ . Yet practical concerns would be well served by the knowledge of what happens in  $Do-All$  when the number of failures is moderate. In particular, it is important to understand the behavior of the best algorithms for the entire range of  $f$ .

The detailed contributions in this work are as follows.

- I. We provide upper bounds (Section 4.1) and matching lower bounds (Section 4.2) that address *all* remaining gaps, hence we give a *complete* analysis of  $Do\text{-}All^O(N, P, f)$  for the *entire* range of  $f$ . The bounds on work  $W$  are<sup>1</sup>

$$\begin{aligned} (a) \quad W &= \{O|\Omega\} \left( N + P \frac{\log P}{\log \log P} \right) \text{ when } f > c \frac{P}{\log P}, \text{ any } c > 0, \\ (b) \quad W &= \{O|\Omega\} \left( N + P \log_{\frac{P}{f}} P \right) \text{ when } f \leq c \frac{P}{\log P}, \text{ any } c > 0. \end{aligned} \quad (1)$$

The lower bounds of course apply to algorithms in weaker models.

The quantity  $\mathcal{Q}_{P,f}$ , defined below and extracted from the bounds (1)(a,b) above, plays an important role in the analysis of complexity of several algorithms.

$$\mathcal{Q}_{P,f} = \begin{cases} P \frac{\log P}{\log \log P} & \text{when } f > c \frac{P}{\log P}, \text{ any } c > 0, \\ P \log_{\frac{P}{f}} P & \text{when } f \leq c \frac{P}{\log P}, \text{ any } c > 0. \end{cases} \quad (2)$$

We use our bounds (1) to derive new bounds for algorithms where the extant analyses do not integrate  $f$  adequately. This is done by analyzing how the work-load balancing is implemented by the algorithms, e.g., by using coordinators or global data-structures. We show the following.

- II. In Section 5.1 we provide new analysis of algorithm AN of Chlebus *et al.* [4] for  $Do\text{-}All$  in the message-passing model with crashes. This algorithm has best known work for moderate number of failures. We show the complete analysis of work  $W$  and message complexity  $M$ :

$$W = O(\log f (N + \mathcal{Q}_{P,f})) \text{ and } M = O(N + \mathcal{Q}_{P,f} + fP).$$

- III. In Section 6.1 we give a *complete* analysis of the work complexity  $W$  of the algorithm of Kanellakis and Shvartsman [17] that solves the  $Do\text{-}All$  (Write-All) problem in synchronous shared-memory systems with processor crashes:

$$W = O(N + \mathcal{Q}_{P,f} \log N).$$

Note that the two algorithms [4][17] are designed for different models and use dissimilar data and control structures, however both algorithms make their load-balancing decisions by gathering global knowledge. By understanding what work is expended on load balancing vs. the inherent work overhead due to the lower bounds (1), we are able to obtain the new results while demonstrating the utility and the generality of our approach.

$Do\text{-}All$  algorithms have been used in developing *simulations* of failure-free algorithms on failure-prone processors, e.g., [21][29]. This is done by iteratively using a  $Do\text{-}All$  algorithm to simulate the steps of the failure-free processors. In this paper we abstract this idea as the *iterative Do-All* problem as follows:

<sup>1</sup> We let “ $\{O|\Omega\}$ ” stand for “ $O$ ” when describing an upper bound and for “ $\Omega$ ” when describing a lower bound. All logarithms are to the base 2 unless explicitly specified otherwise. The expression  $\log X$  stands for  $\max\{1, \log_2 X\}$  for the given  $X$  in the description of complexity results.

The  $r$ -iterative  $Do\text{-}All(N, P, f)$ , or  $r\text{-}Do\text{-}All(N, P, f)$ , is the problem of using  $P$  processors to solve  $r$  instances of  $N$ -task  $Do\text{-}All$  by doing one set of tasks at a time.

The oracle  $r\text{-}Do\text{-}All^O(N, P, f)$  is defined similarly. An obvious solution for this problem is to run a  $Do\text{-}All$  algorithm  $r$  times. If the work complexity of  $Do\text{-}All$  in a given model is  $W_{N,P,f}$ , then the work of  $r\text{-}Do\text{-}All$  is clearly no more than  $r \cdot W_{N,P,f}$ . We present a substantially better analysis:

**IV.** In Section 4.3 we show matching upper and lower bounds on work  $W$  for  $r\text{-}Do\text{-}All^O(N, P, f)$ , where  $f < P \leq N$ , for specific ranges of failures.

$$\begin{aligned} (a) \quad W &= \{O|\Omega\} \left( r \cdot \left( N + P \frac{\log P}{\log \log P} \right) \right) \quad \text{when } f > \frac{Pr}{\log P}, \\ (b) \quad W &= \{O|\Omega\} \left( r \cdot \left( N + P \frac{\log P}{\log \frac{Pr}{f}} \right) \right) \quad \text{when } f \leq \frac{Pr}{\log P}. \end{aligned} \quad (3)$$

We extract the quantity  $\mathcal{R}_{r,P,f}$ , defined below, from the bounds (3)(a,b) above, as it plays an important role in the analysis of iterative  $Do\text{-}All$  algorithms.

$$\mathcal{R}_{r,P,f} = \begin{cases} P \frac{\log P}{\log \log P} & \text{when } f > \frac{Pr}{\log P}, \\ P \frac{\log P}{\log \frac{Pr}{f}} & \text{when } f \leq \frac{Pr}{\log P}. \end{cases} \quad (4)$$

Note that for any  $r$  we have  $\mathcal{Q}_{P,f} \geq \mathcal{R}_{r,P,f}$ , and for the specific range of  $f$  in (4)(b) we have that  $\mathcal{Q}_{P,f} = \omega(\mathcal{R}_{r,P,f})$  with respect to  $r$  (fixed  $P$  and  $f$ ). Thus our bounds (3) are asymptotically better than those obtained by computing the product of  $r$  and the (non-iterated)  $Do\text{-}All$  bounds (1).

**V.** In Section 5.2 we show how to solve  $r\text{-}Do\text{-}All(N, P, f)$  on synchronous message-passing processors with the following work ( $W$ ) and message ( $M$ ) complexity.

$$W = O \left( r \cdot \log \frac{f}{r} \cdot (N + \mathcal{R}_{r,P,f}) \right) \quad \text{and} \quad M = O \left( r \cdot (N + \mathcal{R}_{r,P,f}) + fP \right).$$

**VI.** In Section 6.2 we use  $r\text{-}Do\text{-}All(N, P, f)$  to show that  $P$  processors with crashes can simulate any synchronous  $N$ -processor,  $r$ -time shared-memory algorithm (PRAM) with work:

$$W = O \left( r \cdot (N + \mathcal{R}_{r,P,f} \log N) \right).$$

This last result is strictly better than the previous deterministic bounds for parallel algorithm simulations using the  $Do\text{-}All$  algorithm [17] (the best known to date) and simulation techniques such as [21,29] (due of the the relationship between  $\mathcal{Q}_{P,f}$  and  $\mathcal{R}_{r,P,f}$  as pointed out above).

### 3 Models and Definitions

We define the models, the abstract problem of performing  $N$  tasks in a distributed environment consisting of  $P$  processors that are subject to stop-failures, and the work complexity measure.

**Distributed setting.** We consider a distributed system consisting of  $P$  synchronous processors. We assume that  $P$  is fixed and is known. Each processor has a unique identifier (PID) and the set of PIDs is totally ordered.

**Tasks.** We define a *task* to be a computation that can be performed by any processor in at most one time step; its execution does not depend on any other task. The tasks are also *idempotent*, i.e., executing a task many times and/or concurrently has the same effect as executing the task once. Tasks are uniquely identified by their task identifiers (TIDs) and the set of TIDs is totally ordered. We denote by  $\mathcal{T}$  the set of  $N$  tasks and we assume that  $\mathcal{T}$  is known to all the processors.

**Model of failures.** We assume the *fail-stop* processor model [30]. A processor may crash at any moment during the computation and once crashed it does not restart. We let an omniscient *adversary* impose failures on the system, and we use the term *failure pattern* to denote the set of the events, i.e., crashes, caused by the adversary. A *failure model* is then the set of all failure patterns for a given adversary. For a failure pattern  $F$ , we define the *size*  $f$  of the failure pattern as  $f = |F|$  (the number of failures).

**The Oracle model.** In Section 4 we consider computation where processors are assisted by a deterministic omniscient *oracle*. Any processor may contact the oracle once per step. The introduction of the oracle serves two purposes.

(1) The oracle strengthens the model by providing the processors with any information about the progress of the computation (the oracle cannot predict the future). Thus the lower bounds established for the oracle model also apply to any weaker model, e.g., without an oracle.

(2) The oracle abstracts away any concerns about communication that normally dominate specific message-passing and shared-memory models. This allows for the most general results to be established and it enables us to use these results in the context of specific models by understanding how the information provided by an oracle is simulated in specific algorithms.

**Communication.** In Sections 5 and 6 we deal with message-passing and shared-memory models. For computation in the message-passing model, we assume that there is a known upper bound on message delays. (Communication complexity is defined in Section 5.) When considering computation in the shared-memory model, we assume that reading or writing to a memory cell takes one time unit, and that reads and writes can be concurrent.

**Do-All problems.** We define the *Do-All* problem as follows:

*Do-All: Given a set  $\mathcal{T}$  of  $N$  tasks and  $P$  processors, perform all tasks for any failure pattern in the failure model  $\mathcal{F}$ .*

We let  $\text{Do-All}(N, P, f)$  stand for the *Do-All* problem for  $N$  tasks,  $P$  processors ( $P \leq N$ ), and any pattern of crashes  $F$  such that  $|F| \leq f < P$ . We let  $\text{Do-All}^{\mathcal{O}}(N, P, f)$  stand for the  $\text{Do-All}(N, P, f)$  problem with the oracle. We define the *iterative Do-All* problem as follows:

*Iterative Do-All:* Given any  $r$  sets  $\mathcal{T}_1, \dots, \mathcal{T}_r$  of  $N$  tasks each and  $P$  processors, perform all  $r \cdot N$  tasks, doing one set at a time, for any failure pattern in the failure model  $\mathcal{F}$ .

We denote such  $r$ -iterative *Do-All* by  $r$ -*Do-All*( $N, P, f$ ). The oracle version  $r$ -*Do-All* <sup>$\mathcal{O}$</sup> ( $N, P, f$ ) is defined similarly.

**Measuring efficiency.** We are interested in studying the complexity of *Do-All* measured as *work* (cf. [17, 8, 6]). We assume that it takes a unit of time for a processor to perform a unit of work, and that a single task corresponds to a unit of work. Our definition of *work complexity* is based on the *available processor steps* measure [18]. Let  $\mathcal{F}$  be the adversary model. For a computation subject to a failure pattern  $F$ ,  $F \in \mathcal{F}$ , denote by  $P_i(F)$  the number of processors completing a unit of work in step  $i$  of the computation.

**Definition 1.** Given a problem of size  $N$  and a  $P$ -processor algorithm that solves the problem in the failure model  $\mathcal{F}$ , if the algorithm solves the problem for a pattern  $F$  in  $\mathcal{F}$ , with  $|F| \leq f$ , by time step  $\tau$ , then the work complexity  $W$  of the algorithm is:  $W_{N,P,f} = \max_{F \in \mathcal{F}, |F| \leq f} \left\{ \sum_{i \leq \tau} P_i(F) \right\}$ .

Note that the idling processors consume a unit of work per step even though they do not contribute to the computation. Definition 1 does not depend on the specifics of the target model of computation, e.g., whether it is message-passing or shared-memory. (Communication complexity is defined similarly in Section 5.)

## 4 The Bounds with Perfect Load Balancing

In this section we give the complete analysis of the upper and lower bounds for the  $\text{Do-All}^{\mathcal{O}}(N, P, f)$  and  $r\text{-Do-All}^{\mathcal{O}}(N, P, f)$  problems for the entire range of  $f$  crashes ( $f < P \leq N$ ). (Note: we use the quantities  $\mathcal{Q}_{P,f}$  and  $\mathcal{R}_{r,P,f}$  that are defined in Section 2 in equations (2) and (4) respectively.)

### 4.1 Do-All Upper Bounds

To study the upper bounds for *Do-All* we give an oracle-based algorithm in Figure 1. The oracle tells each processor whether or not all tasks have been performed *Oracle-says*(), and what task to perform next *Oracle-task*() (the correctness of the algorithm is trivial). Thus the oracle performs the termination and load-balancing computation on behalf of the processors.

```

for each processor PID = 1..P begin
  global T[1..N];
  while Oracle-says(PID) = "not done"
    do perform task T[Oracle-task(PID)] od
end.
```

**Fig. 1.** Oracle-based algorithm.



**Lemma 1.** [18,12] *The  $\text{Do-All}^{\mathcal{O}}(N, P, f)$  problem with  $f < P \leq N$  can be solved using work  $W = O\left(N + P \frac{\log P}{\log \log P}\right)$ .*

Note that Lemma 1 does not teach how work depends on the number of crashes  $f$ .

**Lemma 2.** *For any  $c > 0$ ,  $\text{Do-All}^{\mathcal{O}}(N, P, f)$  can be solved for any stop-failure pattern with  $f \leq c \frac{P}{\log P}$  using work  $W = O(N + P + P \log_{P/f} N)$ .*

*Proof.* The proof is based on the proof of Theorem 3.6 of [12].

We now give our main upper-bound result.

**Theorem 1.**  *$\text{Do-All}^{\mathcal{O}}(N, P, f)$  can be solved for any failure pattern using work  $W = O(N + \mathcal{Q}_{P,f})$ .*

*Proof.* This follows directly from Lemmas 1 and 2.

## 4.2 Do-All Lower Bounds

We now show matching lower bounds for  $\text{Do-All}^{\mathcal{O}}(N, P, f)$ . Note that the results in this section hold also for the  $\text{Do-All}(N, P, f)$  problem (without the oracle).

**Lemma 3.** [17,12] *For any algorithm that solves  $\text{Do-All}^{\mathcal{O}}(N, P, f)$  there exists a pattern of  $f$  stop-failures ( $f < P$ ) that results in work  $W = \Omega\left(N + P \frac{\log P}{\log \log P}\right)$ .*

We now define a specific adversarial strategy used to derive our lower bounds. Let  $\text{Alg}$  be any algorithm that solves the  $\text{Do-All}$  problem. Let  $P_i$  be the number of processors remaining at the end of the  $i^{\text{th}}$  step of  $\text{Alg}$  and let  $U_i$  denote the number of tasks that remain to be done at the end of step  $i$ . Initially,  $P_0 = P$  and  $N = U_0$ . Define  $\kappa = \frac{f \log(\frac{P}{f})}{P \log P}$ ,  $0 < \kappa < 1$ .

**Adversary Adv:** At step  $i$  ( $i \geq 1$ ) of  $\text{Alg}$ , the adversary stops processors as follows: Among  $U_{i-1}$  tasks remaining after the step  $i-1$ , the adversary chooses  $U_i = \lfloor \kappa U_{i-1} \rfloor$  tasks with the least number of processors assigned to them and crashes these processors. The adversary continues for as long as  $U_i > 1$ . As soon as  $U_i = 1$  the adversary allows all remaining processors to perform the single remaining task, and  $\text{Alg}$  terminates.

**Lemma 4.** *Given any  $c > 0$  and any algorithm  $\text{Alg}$  that solves  $\text{Do-All}^{\mathcal{O}}(N, P, f)$  for  $P \geq N$ , the adversary  $\text{Adv}$  causes  $f$  stop-failures,  $f \leq c \frac{P}{\log P}$ , and  $W = \Omega(P + P \log_{\frac{P}{f}} N)$ .*

*Proof.* (Sketch.) Let  $\tau$  be the number of iterations caused by adversary  $\text{Adv}$  and  $P_\tau$  be the number of processors at the last iteration of  $\text{Alg}$ . Given the definition of  $\text{Adv}$  (given above), we find that  $\tau \geq \frac{\log N}{\log \frac{P}{f} + \log \log f}$  and  $P_\tau > P - f$ . The result follows by the observation that the work caused by  $\text{Adv}$  is at least  $\tau \cdot P_\tau$ .

**Lemma 5.** *Given any  $c > 0$  and any algorithm  $\text{Alg}$  that solves  $\text{Do-All}^{\mathcal{O}}(N, P, f)$  for  $P \leq N$ , there exists an adversary that causes  $f$  stop-failures,  $f \leq c \frac{P}{\log P}$ , and  $W = \Omega(N + P \log_{\frac{P}{f}} P)$ .*

We now give our main lower-bound result.

**Theorem 2.** *Given any algorithm  $\text{Alg}$  that solves  $\text{Do-All}^{\mathcal{O}}(N, P, f)$  for  $P \leq N$ , there exists an adversary that causes  $W = \Omega(N + \mathcal{Q}_{P,f})$ .*

*Proof.* For the range of failures  $0 < f \leq c \frac{P}{\log P}$  Lemma 5 establishes the bound.

When  $f = c \frac{P}{\log P}$ , the work (per Lemma 5) is  $\Omega\left(N + P \frac{\log P}{\log \log P}\right)$ . For larger  $f$  the adversary establishes this worst case work using the initial  $c \frac{P}{\log P}$  failures.

### 4.3 Iterative Do-All

*Do-All* algorithms have been used in developing simulations of failure-free algorithms on failure-prone processors. This is done by iteratively using a *Do-All* algorithm to simulate the steps of the failure-free processors. We study the *iterative Do-All* problems to understand the complexity implications of iterative use of *Do-All* algorithms.

In principle  $r\text{-Do-All}(N, P, f)$  can be solved by running an algorithm for *Do-All* $(N, P, f)$   $r$  times. If the work of a *Do-All* solution is  $W$ , then the work of the  $r$ -iterative *Do-All* is at most  $r \cdot W$ . However we show that it is possible to obtain a finer result. We refer to each *Do-All* iteration as the *round* of  $r\text{-Do-All}^{\mathcal{O}}(N, P, f)$ .

**Theorem 3.**  *$r\text{-Do-All}^{\mathcal{O}}(N, P, f)$  can be solved with  $W = O(r \cdot (N + \mathcal{R}_{r,P,f}))$ .*

*Proof.* (Sketch) In the case where  $f > \frac{Pr}{\log P}$ , we can have  $\Theta(\frac{P}{\log P})$  processor failures in all  $r$  rounds. From this and Theorem 1, the work is  $O(r \cdot (N + P \frac{\log P}{\log \log P}))$ . Consider the case where  $f \leq \frac{Pr}{\log P}$ . Let  $f_i$  be the number of failures in round  $r_i$ . It is easily shown that the work in every round is  $O(N + P \frac{\log P}{\log f_i})$ . We treat  $f_i$  as a continuous parameter and we compute the first derivative and second derivative w.r.t.  $f_i$ . The first derivative is positive, the second derivative is negative. Hence the first derivative is decreasing (with  $f_i$ ). In this case, given any two  $f_i, f_j$  where  $f_i > f_j$ , the failure pattern obtained by replacing  $f_i$  with  $f_i - \epsilon$  and  $f_j$  by  $f_j + \epsilon$  (where  $\epsilon < (f_i - f_j)/2$ ) results in increased work. This implies that the work is maximized when all  $f_i$ s are equal, specifically when  $f_i = f/r$ . This results to total work of  $O(r \cdot (N + P \frac{\log P}{\log \frac{Pr}{f}}))$ .

**Theorem 4.** *Given any algorithm that solves  $r\text{-Do-All}^{\mathcal{O}}(N, P, f)$  there exists a stop-failure adversary that causes  $W = \Omega(r \cdot (N + \mathcal{R}_{r,P,f}))$ .*

*Proof.* In the case where  $f > \frac{Pr}{\log P}$  the adversary may fail-stop  $\frac{P}{\log P}$  processors in every round of  $r\text{-Do-All}^{\mathcal{O}}(N, P, f)$ . Note that for this adversary  $\Omega(P)$  processors remain alive during the first  $\lceil r/2 \rceil$  rounds. Per Theorem 2 this results in  $\lceil r/2 \rceil \cdot \Omega(N + P \frac{\log P}{\log \log P}) = \Omega(Nr + Pr \frac{\log P}{\log \log P})$  work. In the case where  $f \leq \frac{Pr}{\log P}$

the adversary ideally would kill  $\frac{f}{r}$  processors in every round. It can do that in the case where  $f$  divides  $r$ . If this is not the case, then the adversary kills  $\lceil \frac{f}{r} \rceil$  processors in  $r_A$  rounds and  $\lfloor \frac{f}{r} \rfloor$  in  $r_B$  rounds in such a way that  $r = r_A + r_B$ . Again considering the first half of the rounds and appealing to Theorem 2 results in a  $\Omega(r(N + P \log \frac{rP}{f}))$  lower bound for work. Note that we consider only the case where  $r \leq f$ ; otherwise the work is trivially  $\Omega(rN)$ .

## 5 New Bounds for the Message-Passing Model

In this section we demonstrate the utility of the complexity results under the perfect load-balancing assumption by giving a tight and complete analysis of the algorithm AN [4] and establish new complexity results for the iterative *Do-All* in the message-passing model.

The efficiency of message-passing algorithms is characterized in terms of their work and message complexity. We define message complexity similarly to Definition 1 of work: For a computation subject to a failure pattern  $F$ ,  $F \in \mathcal{F}$ , denote by  $M_i(F)$  the number of point-to-point messages sent during step  $i$  of the computation. For a given problem of size  $N$ , if the computation solves the problem by step  $\tau$  in the presence of the failure pattern  $F$ , where  $|F| \leq F$ , then the message complexity  $M$  is  $M_{N,P,f} = \max_{F \in \mathcal{F}, |F| \leq f} \left\{ \sum_{i \leq \tau} M_i(F) \right\}$ .

### 5.1 Analysis of Algorithm AN

Algorithm AN presented by Chlebus *et al.* [4] uses a multiple-coordinator approach to solve *Do-All*( $N, P, f$ ) on crash-prone synchronous message-passing processors ( $P \leq N$ ). The model assumes that messages incur a known bounded delay and that reliable multicast [15] is available, however messages to/from faulty processors may be lost.

**Description of algorithm AN.** Due to the space limitation, we give a very brief description of the algorithm; for additional details we refer the reader to [4]. Algorithm AN proceeds in a *loop* which is iterated until all the tasks are executed. A single iteration of the loop is called a *phase*. A phase consists of three consecutive *stages*. Each stage consists of three steps. In each stage processors use the first step to receive messages sent in the previous stage, the second step to perform local computation, and the third step to send messages. A processor can be a *coordinator* or a *worker*. A phase may have multiple coordinators. The number of processors that assume the coordinator role is determined by the *martingale principle*: if none of the expected coordinators survive through the entire phase, then the number of coordinators for the next phase is doubled. If at least one coordinator survives in a given phase, then in the next phase there is only one coordinator. A phase that is completed with at least one coordinator alive is called *attended*, otherwise it is called *unattended*.

Processors become coordinators and balance their loads according to each processor's *local view*. A local view contains the set of ids of the processors

assumed to be alive. The local view is partitioned into *layers*. The first layer contains one processor id, the second two ids, the  $i^{\text{th}}$  contains  $2^{i-1}$  ids.

Given a phase, in the first stage, the processors perform a task according to the load balancing rule derived from their local views and report the completion of the task to the coordinators of that phase (determined by their local views). In the second stage, the coordinators gather the reports, they update the knowledge of the done tasks and they multicast this information to the processors that are assumed to be alive. In the last stage, the processors receive the information sent by the coordinators and update their knowledge of done tasks and their local views. Given the full details of the algorithm, it is not difficult to see that the combination of coordinators and local views allows the processors to obtain the information that would be available from the oracle in the algorithm in Figure 1.

It is shown in [4] that the work of algorithm AN is  $W = O((N + N \log N / \log \log N) \log f)$  and its message complexity is  $M = O(N + P \log P / \log \log P + fP)$ .

**New analysis of work complexity.** To assess the work  $W$ , we consider separately all the attended phases and all the unattended phases of the execution. Let  $W_a$  be the part of  $W$  spent during all the attended phases and  $W_u$  be the part of  $W$  spent during all the unattended phases. Hence we have  $W = W_a + W_u$ .

**Lemma 6.** [4] *In any execution of algorithm AN with  $f < P$  we have  $W_a = O(N + P \frac{\log P}{\log \log P})$  and  $W_u = O(W_a \log f)$ .*

We now give the new analysis of algorithm AN.

**Lemma 7.** *In any execution of algorithm AN we have  $W_a = O(N + P \log_{\frac{P}{f}} P)$ , when  $f \leq c \frac{P}{\log P}$ , for any  $c > 0$ .*

**Theorem 5.** *In any execution of algorithm AN we have  $W = O(\log f(N + \mathcal{Q}_{P,f}))$ .*

*Proof.* This follows from Lemmas 6 and 7 and the fact that  $W = W_a + W_u$ .

**Analysis of message complexity.** To assess the message complexity  $M$  we consider separately all the attended phases and all the unattended phases of the execution. Let  $M_a$  be the number of messages sent during all the attended phases and  $M_u$  the number of messages sent during all the unattended phases. Hence we have  $M = M_a + M_u$ .

**Lemma 8.** [4] *In any execution of algorithm AN we have  $M_a = O(W_a)$  and  $M_u = O(fP)$ .*

**Theorem 6.** *In any execution of algorithm AN we have  $M = O(N + \mathcal{Q}_{P,f} + fP)$ .*

*Proof.* It follows from Lemmas 6, 7 and 8 and the fact that  $M = M_a + M_u$ .

## 5.2 Analysis of Message-Passing Iterative Do-All

We now consider the message-passing  $r$ -Do-All( $N, P, f$ ) problem ( $P \leq N$ ).

**Theorem 7.** *The  $r$ -Do-All( $N, P, f$ ) problem can be solved on synchronous crash-prone message-passing processors with work  $W = O(r \cdot \log_{\frac{f}{r}} (N + \mathcal{R}_{r,P,f}))$  and with message complexity  $M = O(r \cdot (N + \mathcal{R}_{r,P,f}) + fP)$ .*

## 6 New Bounds for the Shared-Memory Model

Here we give a new refined analysis of the most work-efficient known *Do-All* algorithm for the shared-memory model, algorithm W [17]. We also establish the complexity results for the iterative *Do-All* and for simulations of synchronous parallel algorithms on crash-prone processors.

### 6.1 Analysis of Algorithm W

Algorithm W solves *Do-All*( $N, P, f$ ) in the shared-memory model (where *Do-All* is better known as *Write-All*). Its work for any pattern of crashes is  $O(N + P \log N \log P / \log \log P)$  for  $P \leq N$  [17]. Note that this bound is conservative, since it does not include  $f$ , the number of crashes.

**Description of the algorithm.** We now give a brief description of the algorithm; for additional details we refer the reader to [18]. Algorithm W is structured as a parallel *loop* through four phases: (W1) a failure detecting phase, (W2) a load rescheduling phase, (W3) a work phase, and (W4) a phase that estimates the progress of the computation, the remaining work and that controls the parallel loop. These phases use full binary trees with  $O(N)$  leaves. The processors traverse the binary trees top-down or bottom-up according to the phase. Each such traversal takes  $O(\log N)$  time (the height of a tree). For a single processor, each iteration of the loop is called a *block-step*; since there are four phases with at most one tree traversal per phase, each block step takes  $O(\log N)$  time.

In algorithm W the trees stored in shared memory serve as the gathering places for global information about the number of active processors, remaining tasks and load balancing. It is not difficult to see that these binary trees indeed provide the information to the processors that would be available from an oracle in the oracle model. The binary tree used in phase W2 to implement load balancing and phase W3 to assess the remaining work is called the *progress tree*.

Here we use the parameterized version of the algorithm with  $P \leq N$  and where the progress tree has  $U = \max\{P, N/\log N\}$  leaves. The tasks are associated with the leaves of this tree, with  $N/U$  tasks per leaf. Note that each block-step still takes time  $O(\log N)$ .

**New complexity analysis.** We now give the work analysis. We charge each processor for each block step it starts, regardless of whether or not the processor completes it or crashes.

**Lemma 9.** [18] *For any failure pattern with  $f < P$ , the number of block-steps required by the  $P$ -processor algorithm W with  $U$  leaves in the progress tree is  $B = O\left(U + P \frac{\log P}{\log \log P}\right)$ .*

**Lemma 10.** *For any failure pattern with  $f \leq c \frac{P}{\log P}$  (for any  $c > 0$ ), the number of block-steps required by the  $P$ -processor algorithm W with  $U$  leaves in the progress tree is  $B = O(U + P \log_{\frac{P}{f}} P)$ .*

**Theorem 8.** *Algorithm W solves *Do-All*( $N, P, f$ ) with work  $O(N + \mathcal{Q}_{P,f} \log N)$ .*

## 6.2 Iterative Do-All and Parallel Algorithm Simulations

We now consider the complexity of shared-memory  $r$ -Do-All( $N, P, f$ ) and of PRAM simulations.

**Theorem 9.** *The  $r$ -Do-All( $N, P, f$ ) problem can be solved on  $P$  crash-prone processors ( $P \leq N$ ), using shared memory, with work  $W = O(r \cdot (N + \mathcal{R}_{r,P,f} \log N))$ .*

**Theorem 10.** *Any synchronous  $N$ -processor,  $r$ -time shared-memory parallel algorithm (PRAM) can be simulated on  $P$  crash-prone synchronous processors with work  $O(r \cdot (N + \mathcal{R}_{r,P,f} \log N))$ .*

## 7 Conclusions

In this paper we give the first complete analysis of the *Do-All* problem under the perfect load-balancing assumption. We introduce and analyze the *iterative Do-All* problem that models repeated use of *Do-All* algorithms, such as found in algorithm simulations and transformations. A unique contribution of our analyses is that they precisely describe the effect of crash failures on the work of the computation. We demonstrate the utility of the analyses obtained with the perfect load-balancing assumption by using them to analyze message-passing and shared-memory algorithms and simulations that attempt to balance the loads among the processors.

## References

1. Aumann, Y., Rabin, M.O.: Clock Construction in Fully Asynchronous Parallel Systems and PRAM Simulation. 33rd IEEE Symp. on Foundations of Computer Science (1993) 147–156.
2. Anderson, R.J., Woll, H.: Algorithms for the Certified Write All Problem. SIAM Journal of Computing, Vol. 26 **5** (1997) 1277–1283.
3. Buss, J., Kanellakis, P.C., Ragde, P., Shvartsman, A.A.: Parallel Algorithms with Processor Failures and Delays. Journal of Algorithms, Vol. 20 (1996) 45–86.
4. Chlebus, B.S., De Prisco, R., Shvartsman, A.A.: Performing Tasks on Restartable Message-Passing Processors. Distributed Computing, Vol. 14 **1** (2001) 49–64.
5. Dasgupta, P., Kadem, Z., Rabin, M.: Parallel Processing on Networks of Workstation: A Fault-Tolerant, High Performance Approach. International Conference on Distributed Computer Systems (1995) 467–474.
6. De Prisco, R., Mayer, A., Yung, M.: Time-Optimal Message-Efficient Work Performance in the Presence of Faults. 13th ACM Symposium on Principles of Distributed Computing (1994) 161–172.
7. Dolev, S., Segala, R., Shvartsman, A.: Dynamic Load Balancing with Group Communication. 6th International Colloquium on Structural Information and Communication Complexity (1999) 111–125.
8. Dwork, C., Halpern, J., Waarts, O.: Performing Work Efficiently in the Presence of Faults. SIAM J. on Computing, Vol. 27 **5** (1998) 1457–1491.
9. Fischer, M.J., Lynch, N.A., Paterson, M.S.: Impossibility of Distributed Consensus with one Faulty Process. Journal of the ACM, Vol. 32 **2** (1985) 374–382.

10. Galil, Z., Mayer, A., Yung, M.: Resolving Message Complexity of Byzantine Agreement and Beyond. 36th IEEE Symp. on Foundations of Comp. Sc. (1995) 724–733.
11. Georgiou, C., Shvartsman, A.: Cooperative Computing with Fragmentable and Mergeable Groups. 7th International Colloquium on Structural Information and Communication Complexity (2000) 141–156.
12. Georgiou, C., Russell, A., Shvartsman, A.: The Complexity of Distributed Cooperation in the Presence of Failures. 4th International Conference on Principles of Distributed Systems (2000) 245–264.
13. Georgiou, C., Russell, A., Shvartsman, A.: The Complexity of Synchronous Iterative Do-All with Crashes. <http://www.engr.uconn.edu/~acr/Papers/faults.ps>.
14. Groote, J.F., Hesselink, W.H., Mauw, S., Vermeulen, R.: An Algorithm for the Asynchronous Write-All Problem Based on Process Collision. Distributed Computing (2001).
15. Hadzilacos, V., Toueg, S.: Fault-Tolerant Broadcasts and Related Problems. Distributed Computing, 2nd Ed., Addison-Wesley and ACM Press (1993).
16. Hesselink, W.H., Groote, J.F.: Waitfree Distributed Memory Management by Create, and Read Until Deletion (CRUD). Technical report SEN-R9811, CWI, Amsterdam (1998).
17. Kanellakis, P.C., Shvartsman, A.A.: Efficient Parallel Algorithms Can Be Made Robust. Distributed Computing, Vol. 5 (1992) 201–217.
18. Kanellakis, P.C., Shvartsman, A.A.: Fault-Tolerant Parallel Computation. Kluwer Academic Publishers (1997) ISBN 0-7923-9922-6.
19. Kadem, Z.M., Palem, K.V., Raghunathan, A., Spirakis, P.: Combining Tentative and Definite Executions for Dependable Parallel Computing. 23d ACM Symposium on Theory of Computing (1991) 381–390.
20. Kadem, Z.M., Palem, K.V., Rabin, M.O., Raghunathan, A.: Efficient Program Transformations for Resilient Parallel Computation via Randomization. 24th ACM Symp. on Theory of Computing (1992) 306–318.
21. Kadem, Z.M., Palem, K.V., Spirakis, P.: Efficient Robust Parallel Computations. 22nd ACM Symp. on Theory of Computing (1990) 138–148.
22. Lamport, L., Lynch, N.A.: Distributed Computing: Models and Methods. Handbook of Theoretical Computer Science, Vol. 1, North-Holland (1990).
23. Lamport, L., Shostak, R., Pease, M.: The Byzantine Generals Problem. ACM TOPLAS, Vol. 4 **3** (1982) 382–401.
24. Malewicz, G.G., Russell, A., Shvartsman, A.A.: Distributed Cooperation in the Absence of Communication. 14th International Symposium on Distributed Computing (2000) 119–133.
25. Martel, C., Subramonian, R.: On the Complexity of Certified Write-All Algorithms. Journal of Algorithms, Vol. 16 **3** (1994) 361–387.
26. Martel, C., Park, A., Subramonian, R.: Work-Optimal Asynchronous Algorithms for Shared Memory Parallel Computers. SIAM Journal on Computing, Vol. 21 (1992) 1070–1099.
27. Martel, C., Subramonian, R., Park, A.: Asynchronous PRAMs are (Almost) as Good as Synchronous PRAMs. 32d IEEE Symp. on Foundations of Computer Science (1990) 590–599.
28. Pease, M., Shostak, R., Lamport, L.: Reaching Agreement in the Presence of Faults. Journal of the ACM, Vol. 27 **2** (1980) 228–234.
29. Shvartsman, A.A.: Achieving Optimal CRCW PRAM Fault-Tolerance. Information Processing Letters, Vol. 39 **2** (1991) 59–66.
30. Schlichting, R.D., Schneider, F.B.: Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems. TOCS 1, Vol. 3 (1983) 222–238.

# Mobile Search for a Black Hole in an Anonymous Ring

Stefan Dobrev<sup>1</sup>, Paola Flocchini<sup>2</sup>, Giuseppe Prencipe<sup>3</sup>, and Nicola Santoro<sup>4</sup>

<sup>1</sup> Slovak Academy of Sciences, stefan@ifi.savba.sk

<sup>2</sup> University of Ottawa, flocchin@site.uottawa.ca

<sup>3</sup> Università di Pisa, prencipe@di.unipi.it

<sup>4</sup> Carleton University, santoro@scs.carleton.ca

**Abstract.** We address the problem of *mobile agents searching a ring network* for a highly harmful item, a *black hole*, a stationary process destroying visiting agents upon their arrival. No observable trace of such a destruction will be evident. The location of the black hole is not known; the task is to unambiguously determine and report the location of the black hole. We answer some natural computational questions: *How many agents are needed to locate the black hole in the ring ? How many suffice? What a-priori knowledge is required?* as well as complexity questions, such as: *With how many moves can the agents do it ? How long does it take ?*

**Keywords:** Mobile Agents, Distributed Computing, Ring Network, Hazardous Search.

## 1 Introduction

The most widespread use of autonomous mobile agents in network environments, from the World-Wide-Web to the Data Grid, is clearly to *search*, i.e., to locate some required “item” (e.g., information, resource, ...) in the environment. This process is started with the specification of what must be found and ends with the reporting of where it is located.

The proposed solutions integrate their algorithmic strategies with an exploitation of the capabilities of the network environment; so, not surprising, they are varied in nature, style, applicability and performance (e.g., see [3,4,11,14,16]). They do however share the same assumption about the “item” to be located by the agents: it poses no danger, it is *harmless*.

This assumption unfortunately does not always hold: the item could be a local program which severely damages visiting agents. In fact, protecting an agent from “host attacks” (i.e., harmful items stored at the visited site) has become a problem almost as pressing as protecting a host (i.e., a site) from an agent attack (e.g., see [17,18]). Still, this problem has not been taken into account so far by any of the existing solutions.

In this paper we address the problem of searching for a highly harmful item whose existence we are aware of, but whose whereabouts are unknown. The item



is a stationary process which disposes of visiting agents upon their arrival; no observable trace of such a destruction will be evident. Because of its nature, we shall call such an item a *black hole*.

The task is to unambiguously determine and report the location of the black hole (following this phase, a “rescue” activity would conceivably be initiated to deal with such a destructive process). In the distributed computing literature, there have been many studies on computing in presence of undetectable faulty components (e.g., [5,10] which can be rephrased in terms of computations in presence of black holes). However, as mentioned, this problem has never been investigated before. We are interested in understanding the basic algorithmic limitations and factors. The setting we consider is the simplest symmetric topology: the anonymous ring (i.e., a loop network of identical nodes). In this setting operate mobile agents: the agents have limited computing capabilities and bounded storage<sup>1</sup>, obey the same set of behavioral rules (the “protocol”), and can move from node to neighboring node. We make no assumptions on the amount of time required by an agent’s actions (e.g., computation, movement, etc) except that it is finite; thus, the agents are *asynchronous*. Each node has a bounded amount of storage, called *whiteboard*;  $O(\log n)$  bits suffice for all our algorithms. Agents communicate by reading from and writing on the whiteboards; access to a whiteboard is done in mutual exclusion.

Some basic computational questions naturally and immediately arise, such as: *How many agents are needed to locate the black hole ? How many suffice ? What a-priori knowledge is required ?* as well as complexity questions, such as: *With how many moves can the agents do it ? How long does it take ?*

In this paper, we provide some definite answers to each of these questions. Some answers follow from simple facts. For example, if the existence of the black hole is a possibility but not a certainty, it is impossible<sup>2</sup> to resolve this ambiguity. Similarly, if the ring size  $n$  is not known, then the black-hole search problem can not be solved. Hence,  $n$  must be known. Another fact is that at least two agents are needed to solve the problem.

A more interesting fact is that if the agents are *co-located* (i.e., start from the same node) and *anonymous* (i.e., do not have distinct labels), then the problem is unsolvable. Therefore, to find the black hole, co-located agents must be *distinct* (i.e., have different labels); conversely, anonymous agents must be *dispersed* (i.e., start from different nodes). In this paper we consider both settings.

We first consider *distinct co-located* agents. We prove that *two* such agents are both *necessary and sufficient* to locate the black hole. Sufficiency is proved constructively: we present a distributed algorithm which allows locating the black hole using only two agents. This algorithm is *optimal*, within a factor of two, also in terms of the *amount of moves* performed by the two agents. In fact, we show the stronger result that  $(n - 1) \log(n - 1) + O(n)$  moves are needed regardless of the number of co-located agents, and that with our algorithm two agents can solve the problem with no more than  $2n \log n + O(n)$  moves.

<sup>1</sup>  $O(\log n)$  bits suffice for all our algorithms.

<sup>2</sup> i.e., no deterministic protocol exists which always correctly terminates.

We also focus on the minimal *amount of time* spent by co-located agents to locate the black hole. We easily show that  $2n - 4$  (ideal) time units are needed, regardless of the number of agents; we then describe how to achieve such a time bound using only  $n - 1$  agents. We generalize this technique and establish a *general trade-off* between time and number of agents.

We then consider *anonymous dispersed* agents. We prove that, *two* anonymous dispersed agents are both *necessary and sufficient* to locate the black hole if the ring is *oriented*. Also in this case the proof of sufficiency is constructive: we present an algorithm which, when executed by two or more anonymous agents dispersed in an unoriented ring, allows finding the black hole with  $O(n \log n)$  moves. This algorithm is *optimal* in terms of *number of moves*; in fact, we prove that any solution with  $k$  anonymous dispersed agents requires  $\Omega(n \log(n - k))$  moves, provided  $k$  is known; if  $k$  is unknown,  $\Omega(n \log n)$  moves are always required.

We also show that *three* anonymous dispersed agents are *necessary and suffice* if the ring is *unoriented*. Sufficiency follows constructively from the result for oriented rings. Due to space restrictions, some of the proofs will be omitted.

## 2 Basic Results and Lower Bound

### 2.1 Notation and Assumptions

The network environment is a set  $A$  of *asynchronous* mobile agents in a ring  $\mathcal{R}$  of  $n$  *anonymous* (i.e., unlabeled<sup>3</sup>) nodes. The size  $n$  of  $\mathcal{R}$  is known to the agents; the number of agents  $|A| = k \geq 2$  might not be a priori known. The agents can move from node to neighboring node in  $\mathcal{R}$ , have computing capabilities and bounded storage, obey the same set of behavioral rules (the “protocol”), and all their actions (e.g., computation, movement, etc) take a finite but otherwise unpredictable amount of time. Each node has two ports, labelled *left* and *right*; if this labelling is globally consistent, the ring will be said to be *oriented*, *unoriented* otherwise. Each node has a bounded amount of storage, called *whiteboard*;  $O(\log n)$  bits suffice for all our algorithms. Agents communicate by reading from and writing on the whiteboards; access to a whiteboard is done in mutual exclusion. A *black hole* is a stationary process located at a node, which destroys any agent arriving at that node; no observable trace of such a destruction will be evident to the other agents.

The location of the black hole is unknown to the agents. The *Black-Hole Search* (BHS) problem is to *find the location of the black hole*. More precisely, BHS is solved if at least one agent survives, and all surviving agents know the location of the black hole (*explicit termination*). Notice that our lower bounds are established requiring only that at least one surviving agent knows the location of the black hole (the difference is only  $O(N)$  moves/time).

First of all notice that, because of the asynchrony of the agents, we have that:

<sup>3</sup> Alternatively, they all have the same label.

**Fact 1.** *It is impossible to distinguish between slow links and a black hole.*

This simple fact has several important consequences; in particular:

**Corollary 1.**

1. *It is impossible to determine (using explicit termination) whether or not there is a black hole in the ring.*
2. *Let the existence of the black hole in  $\mathcal{R}$  be common knowledge. It is impossible to find the black hole if the size of the ring is not known.*

Thus, we assume that both the existence of the black hole and the size  $n$  of the ring are common knowledge to the agents. The agents are said to be *co-located* if they all start from the same node; if they initially are in different nodes, they are said to be *dispersed*.

**Fact 2.** *Anonymous agents starting at the same node collectively behave as one agent.*

**Corollary 2.** *It is impossible to find the black hole if the agents are both co-located and anonymous.*

Thus, we assume that if the agents are initially placed in the same node, they have distinct identities; on the other hand, if they start from different locations there is at most one agent starting at any given node. Finally, observe the obvious fact that if there is only one agent the BHS problem is unsolvable; that is

**Fact 3.** *At least two agents are needed to locate the black hole.*

Thus, we assume that there are at least two agents. Let us now introduce the complexity measure used in the paper. Our main measures of complexity are the *number of agents*, called *size*, and the total *number of moves* performed by the agents, which we shall call *cost*. We will also consider the amount of *time* elapsed until termination. Since the agents are asynchronous, “real” time cannot be measured. We will use the traditional measure of *ideal time* (i.e., assuming synchronous execution where a move can be made in one time unit); sometimes we will also consider *bounded delay* (i.e., assuming an execution where a move requires at most one time unit), and *causal time* (i.e., the length of the longest, over all possible executions, chain of causally related moves). In the following, unless otherwise specified, “time” complexity is “ideal time” complexity.

## 2.2 Cautious Walk

At any time during the search for the black hole, the ports (corresponding to the incident links) of a node can be classified as (a) *unexplored* – if no agent has moved across this port, (b) *safe* – if an agent arrived via this port or (c) *active* – if an agent departed via this port, but no agent has arrived via it.

It is always possible to avoid sending agents over *active* links using a technique we shall call *cautious walk*: when an agent moves from node  $u$  to  $v$  via an *unexplored* port (turning it into *active*), it must immediately return to  $u$  (making the port *safe*), and only then go back to  $v$  to resume its execution; an agent

needing to move from  $u$  to  $v$  via an *active* port must wait until the port becomes *safe*. In the following, by the expression *moving cautiously* we will mean moving using cautious walk. *Cautious walk* reduces the number of agents that may enter the black hole in a ring to 2 (i.e., the degree of the node containing the black hole). Note that this technique can be used in any asynchronous algorithm  $\mathcal{A}$ , at a cost of  $O(n)$  additional moves, with minimal consequences:

**Lemma 1 ([8]).** *Let  $\mathcal{A}'$  be the algorithm obtained from  $\mathcal{A}$  by enforcing cautious walk. For every execution  $\mathcal{E}'$  of  $\mathcal{A}'$  there exists a corresponding execution  $\mathcal{E}$  of  $\mathcal{A}$  such that  $\mathcal{E}$  is obtained from  $\mathcal{E}'$  by deleting only the additional moves due to cautious walk.*

Let us remark that cautious walk is a general technique that can be used in any topology; furthermore, it has been shown that *every* black-hole location algorithm for two agents *must* use cautious walk [8].

### 2.3 Lower Bound on Moves

In this section we consider the minimum number of moves required to solve the problem. The existence of an asymptotic  $\Omega(n \log n)$  bound can be proven by carefully adapting and modifying the (rather complex) proof of the result of [10] on rings with a faulty link. In the following, using a substantially different argument, we are able to obtain directly a more precise (not only asymptotic) bound, with a simpler proof. In fact we show that, regardless of the setting (i.e., collocation or dispersal) and of the number of agents employed,  $(n - 1) \log(n - 1) + O(n)$  moves are required.

In the following, we will denote by  $\mathcal{E}^t$  and  $\mathcal{U}^t$  the explored and unexplored area at time  $t$ , respectively. Moreover,  $z^t$  denotes the central node of  $\mathcal{E}^t$ ; that is, given  $x^t$  and  $y^t$ , the two *border nodes* in  $\mathcal{E}^t$  that connect  $\mathcal{E}^t$  to  $\mathcal{U}^t$ ,  $z^t$  is the node in  $\mathcal{E}^t$  at distance  $\lceil |\mathcal{E}^t|/2 \rceil - 1$  from  $x^t$ .

**Definition 1.** *A causal chain from a node  $v_p$  to a node  $v_q$  has been executed at time  $t$ , if  $\exists d \in \mathbb{N}, \exists u_1, u_2, \dots, u_d \in V$  and times  $t_1, t'_1, t_2, t'_2, \dots, t_d, t'_d$  such that*

- $t < t_1 < t'_1 < t_2 < t'_2 < \dots < t_d < t'_d$ ,
- $v_p = u_1, v_q = u_d$  and  $\forall i \in \{1, 2, \dots, d - 1\} : u_i$  is a neighbor of  $u_{i+1}$ , and
- $\forall i \in \{1, 2, \dots, d - 1\}$  at time  $t_i$  an agent moves from node  $u_i$  to node  $u_{i+1}$  and reaches  $u_{i+1}$  at time  $t'_i$ .

**Lemma 2.** *Let  $|\mathcal{U}^t| > 2$  at a given time  $t \geq 0$ , and  $k \geq 2$ .*

1. *Within finite time, at least two agents will leave the explored area  $\mathcal{E}^t$  in different directions.*
2. *A finite time after they have left  $\mathcal{E}^t$ , say at  $t' > t$ , a causal chain is executed from one of the two border nodes of  $\mathcal{E}_{t'}$  to  $z_t$ .*

**Theorem 1.** *At least  $(n - 1) \log(n - 1) + O(n)$  moves are needed to find a black hole in a ring, regardless of the number of agents.*

### 3 Co-located Distinct Agents

In this section we consider the case when all agents are co-located but distinct; i.e., they start at the same node, called the *home base*, and have distinct identities. The distinct labels of the agents allows any tie to be deterministically broken. As a consequence, if the ring is unoriented, the agents can initially agree on the directions of the ring. Thus, in the rest of this section, we assume w.l.g. that the ring is oriented.

Let  $0, 1, \dots, n-1$  be the nodes of the ring in clockwise direction ( $0, -1, \dots, -(n-1)$  in counter-clockwise direction) and, without loss of generality, let us assume that node 0 is the home base.

#### 3.1 Agent-Optimal Solution

At least two agents are needed to locate the black hole (Fact 3). We now consider the situation when there are *exactly* two agents,  $l$  and  $r$ , in the system, and they are co-located.

The algorithm proceeds in phases. Let  $E_i$  and  $U_i$  denote the explored and unexplored nodes in phase  $i$ , respectively. Clearly,  $E_i$  and  $U_i$  partition the ring into two connected subgraphs, with the black hole located somewhere in  $U_i$ .

#### Algorithm 1 (Two Agents)

Start with round number  $i = 1$ ,  $E_1 = \{0\}$ , and  $U_1 = \{1, 2, \dots, n-1\}$ .

1. Divide  $U_i$  into two continuous disjoint parts  $U_i^l$  and  $U_i^r$  of almost equal sizes. Since  $U_i$  is a path, this is always possible. (We may assume  $U_i^l$  is to the left of 0 while  $U_i^r$  is to the right.)
2. Let agents  $l$  and  $r$  explore (using *Cautious Walk*)  $U_i^l$  and  $U_i^r$ , respectively. Note that, since both of them are within  $E_i$  and since  $U_i$  is divided into two continuous parts, the agents can safely reach the parts they have to explore.
3. Since  $U_i^l$  and  $U_i^r$  are disjoint, at most one of them contains the black hole; hence, one of the agents (w.l.g. assume  $r$ ) successfully completes step 2. Agent  $r$  then moves across  $E_i$  and follows the *safe* ports of  $U_i^l$  until it comes to the node  $w$  from which there is no *safe* port leading to the left.
4. Denote by  $U_{i+1}$  the remaining unexplored area. (All nodes to the right of  $w$ , up to the last node of  $U_i^r$  explored by  $r$ , are now explored – they form  $E_{i+1}$ .) If  $|U_{i+1}| = 1$ , agent  $r$  knows that the black hole is in the single unexplored node and **terminates**. Otherwise  $U_{i+1}$  is divided into  $U_{i+1}^l$  and  $U_{i+1}^r$  as in step 1. Agent  $r$  leaves on the whiteboard of  $w$  a message for  $l$  indicating the two areas  $U_{i+1}^l$  and  $U_{i+1}^r$ . Note that  $O(\log n)$  bits are sufficient to code this message.
5. Agent  $r$  traverses  $E_{i+1}$  and starts exploring  $U_{i+1}^r$ . (Proceeds to the next round – increment  $i$  and go to step 2 ...)
6. When (if)  $l$  returns to  $w$ , it finds the message and starts exploring  $U_{i+1}^l$ . (Proceeds to the next round – increment  $i$  and go to step 2 ...)  $\blacktriangleleft$

**Theorem 2.** *Two agents can find the black hole performing  $2n \log n + O(n)$  moves (in time  $2n \log n + O(n)$ ).*

From Fact 3 and Theorems 1 and 2 it follows that

**Corollary 3.** *Algorithm 1 is size-optimal and cost-optimal.*

### 3.2 More Than Two Agents: Improving the Time

In this section we study the effects of having  $k > 2$  agents in the home base.

We know that increasing  $k$  will not result in a decrease of the total number of moves; in fact, the lower bound of Theorem 1 is independent of the number of agents and is already achieved, within a factor of two, by  $k = 2$ . However, the availability of more agents can be exploited to improve the *time complexity* of locating the black hole.

The following theorem shows a simple lower bound on the time needed to find the black hole, regardless of the number of agents in the system.

**Theorem 3.** *In the worst case,  $2n - 4$  time units are needed to find the black hole, regardless of the number of agents available.*

We now show that the lower bound can be achieved employing  $n - 1$  agents. Let  $r_1 \dots r_{(n-1)}$  be the  $n - 1$  agents.

#### Algorithm 2 ( $n - 1$ Agents)

Each agent  $r_i$  is assigned a location  $i + 1$ ; its task is to verify whether that is the location of the black hole. It does so in two steps, executed independently of the other agents.

Step 1: It first goes to node  $i$  in clockwise direction and, if successful, returns to the home base (phase 1).

Step 2: It then goes in counter clockwise direction to node  $-(n - i - 2)$  and, if successful, returns to the home base: the assigned location is where the black hole resides.

Clearly, only one agent will be able to complete both steps, while the other  $n - 2$  will be destroyed by the black hole.  $\blacktriangleleft \blacktriangleright$

**Theorem 4.** *The black hole can be found in time  $2n - 4$  by  $n - 1$  agents starting from the same node.*

Thus, by Theorems 3 and 4, it follows that

**Corollary 4.** *Algorithm 2 is time-optimal.*

We now show how to employ the idea used for the time-optimal algorithm to obtain a trade-off between the number of agents employed and the time needed to find the black hole. Let  $q$  ( $1 \leq q \leq \log n$ ) be the trade-off parameter.

**Algorithm 3 (Time-Size Tradeoff )**

Two agents (called *explorers*) are arbitrarily chosen; their task is to mark all the safe ports before the black hole. They do so by leaving the home base in opposite directions, moving cautiously; each will continue until it is destroyed by the black hole.

The other agents start their algorithm in pipeline with the two explorers, always leaving from safe ports. The algorithm proceeds in  $q$  rounds.

In each round  $n^{1/q} - 1$  agents ( $r_1 \dots r_{n^{1/q}-1}$ ) follow an algorithm similar to Algorithm 2 to reduce the size of the unexplored area by a factor of  $n^{1/q}$ . The unexplored area is in fact divided into  $n^{1/q}$  segments  $S_1, S_2, \dots, S_{n^{1/q}}$  of almost equal size (e.g., at the first phase the segment  $S_i$  is  $(i-1)n^{(q-1)/q} + 1, \dots, in^{(q-1)/q}$ ). Agent  $r_i$  verifies the guess that the black hole belongs to segment  $S_i$  by checking the nodes around  $S_i$  (first the right one, then the left one).

Clearly only one agent, say  $r_i$ , will be able to locate the segment containing the black hole. When  $r_i$  verifies its guess, arriving to the node to the left of  $S_i$ , the agents  $r_j$  with  $j < i$  are trying to enter  $S_i$  from the left, while the agents  $r_j$  for  $j > i$  are still trying to enter  $S_i$  from the right. To use these agents in the next round,  $r_i$  has to “wake them up”: before returning to the home base,  $r_i$  moves left (possibly entering  $S_i$ ) up to the last safe port, awakening all  $r_j$  with  $j < i$ , so that they can correctly proceed to the next round.

The process is repeated until the black hole is located in round  $q$ . ◀▶

Notice that, except for the two exploring agents, all agents survive.

**Theorem 5.** *Let  $1 \leq q \leq \log n$ . The black hole can be found using  $n^{1/q} + 1$  agents in time  $2(q+1)n - o(n)$ .*

## 4 Dispersed Anonymous Agents

In this section we examine the case when the agents are anonymous but dispersed (i.e., initially there is at most one agent at any given location). The number  $k$  of agents is not known a priori.

### 4.1 Basic Properties and Lower Bounds

A simple but important property is that, although anonymous, the agents can uniquely identify each other by means of purely local names. This is easily achieved as follows. Each agent  $a$  will think of the nodes as numbered with consecutive integers in the clockwise direction, with its starting node (its “homebase”) as node 0. Then, when moving, agent  $a$  will keep track of the relative distance  $d_a$  from the homebase: adding +1 when moving clockwise, and -1 otherwise. Thus, when  $a$  encounters at the node (at distance)  $d_a = -3$  an agent  $b$  which is at distance  $d_b = +2$  from its own homebase,  $a$  is able to unambiguously determine that  $b$  is the unique agent whose homebase is node -5 (in  $a$ ’s view of the ring).

**Lemma 3.** *Each agent can distinguish and recognize all other agents.*

Another simple but important property is that, unlike the case of co-located agents, with dispersed agents there is a major difference between oriented and unoriented rings. In fact, if the ring is unoriented, two agents no longer suffice to solve the problem: they could be located in the nodes next to the black hole, and both made to move towards it. In other words,

**Fact 4.** *At least three dispersed agents are needed to locate the black hole in the unoriented ring.*

Thus, when dealing with the unoriented ring, we assume that there are at least three dispersed agents.

We now establish a lower bound (that we will prove to be tight in the next section) on the cost for locating the black hole; the lower bound is established for oriented rings and, thus, applies also to the unoriented case.

**Theorem 6.** *The cost of locating the black hole in oriented rings is at least  $\Omega(n \log n)$ .*

We now consider the case when every agent is endowed with a priori knowledge of  $k$ . This additional knowledge would provide little relief, as indicated by the following lower bound.

**Theorem 7.** *If  $k$  is known a priori to the agents, the cost of locating the black hole in oriented ring is  $\Omega(n \log(n - k))$ .*

The proof of Theorem 6 considers a worst-case scenario: an adversarial placement of both the black hole and the agents in the ring. So, one last question is whether, knowing  $k$  we could fare substantially better under a (blind but) favorable placement of the agents in the ring; i.e., assuming that  $k$  is known a priori and that we can place the agents, leaving to the adversary only the placement of the black hole. Also in this case, the answer is substantially negative. In fact, the application of the proof technique of Theorem 1 (with the initial explored region set to be the smallest connected region containing all agents, which is clearly of size at most  $n - n/k$ ) yields a lower bound of  $\Omega(n \log(n/k)) = \Omega(n(\log n - \log k))$ , which, for reasonably small  $k$ , is still  $\Omega(n \log n)$ .

## 4.2 Oriented Rings: A Cost-Optimal Algorithm

In this section we describe a cost-optimal algorithm for the oriented ring where  $k \geq 2$  anonymous agents are dispersed. The algorithm is composed of three distinct parts: pairing, elimination, and resolution.

The basic idea is to first form pairs of agents and then have the pairs search for the black hole.



**Algorithm 4 (Pairing)**

1. Move along the ring clockwise using cautious walk, marking (direction and distance to the starting node) the visited nodes, until arriving to a node visited by another agent.
2. Chase that agent until you come to a) a node visited by two other agents or b) the last safe node marked by the agent you are chasing  
     Case a) **Terminate** with status *alone*.  
     Case b) Form a pair: Leave a mark *Join me* and **terminate** with status *paired-left*
3. When, during your cautious walk, you encounter the mark *Join me*, clear this mark and **terminate** with status *paired-right*.
4. If you meet a paired agent, **terminate** with status *alone*. ◁▷

The agents with status *paired-* will then execute the algorithm to locate the black hole. The agents terminating with status *alone* will be passive in the remainder of the computation.

**Lemma 4.** *At least one pair is formed during the pairing phase. The pairing phase lasts at most  $3n - 6$  time units, its cost is at most  $4n - 7$ .*

Note that, if the pairing algorithm starts with  $k$  agents, any number of pairs between 1 and  $\lfloor k/2 \rfloor$  can be formed, depending on the timing. For example  $\lfloor k/2 \rfloor$  pairs are formed when the “even” (as counting to the left from the black hole) agents are very slow, and the “odd” agents are fast and catch their right neighbors.

Since agents can distinguish themselves using local names based on their starting nodes (Lemma 3), also the pairs can be given local names, based on the node where the pair was formed (the “homebase”). This allows a pair of agents to ignore all other agents. Using this fact, a straightforward solution consists of having each pair independently execute the location algorithm for two agents (Algorithm 1). This however will yield an overall  $O(n^2 \log n)$  worst-case cost.

To reduce the cost, the number of active pairs must be effectively reduced. The reduction is done in a process, called *elimination*, closely resembling leader election. In this process, the number of homebases (and thus pairs) is reduced to at most two.

The two agents in the pair formed at node  $v$  will be denoted by  $r_v$  and  $l_v$ , and referred to as the right and the left agent, respectively;  $v$  will be their homebase.

**Algorithm 5 (Elimination)**

The computation proceeds in logical rounds. In each round, the left agent  $l_v$  cautiously moves to the left until it is destroyed by the black hole (case 0), or it reaches a homebase  $u$  with higher (case 1) or equal (case 2) round number. In case (1),  $l_v$  returns to  $v$  which it marks *Dead*. In case (2),  $l_v$  marks  $u$  as *Dead* and returns to  $v$ ; if  $v$  is not marked *Dead*, it is promoted to the next round.

Similarly, agent  $r_v^*$  cautiously moves to the right until it finds (if it is not destroyed by the black hole) the first homebase  $u$  in equal or higher round; it then returns back to  $v$ . If the current level of  $v$  (its level could have risen during the travel of  $r_v^*$ ) is not higher then the level of  $u$ ,  $v$  is marked *Dead*; otherwise, if  $v$  is not marked *Dead*,  $r_v^*$  travels again to the right (it is now in a higher round).

To prevent both agents of a pair entering the black hole, both  $l_v^*$  and  $r_v^*$  maintain a counter and travel to distance at most  $\lfloor (n-1)/2 \rfloor$ . If one of them has traveled such a distance without finding another homebase with the same or higher round, it returns back to  $v$ , and  $v$  is marked *Selected*.  $\langle \rangle$

The rule of case 1 renders stronger a homebase (and, thus, a pair) in a higher logical round; ties are resolved giving priority to the right node (case 2 and the handling of the right agent). This approach will eventually produce either one or two *Selected* homebases. If an agent returns to a homebase marked *Dead*, it stops any further execution. When a homebase has been marked *Selected*, the corresponding agents will then start the resolution part of the algorithm by executing Algorithm 1, and locating the black hole. Note that, for each of the two pairs, the execution is started by a single agent; the other agent either has been destroyed by the black hole or will join in the execution upon its return to the homebase. Summarizing, the overall algorithm is structured as follows.

#### Algorithm 6 (Overall)

1. Form pairs of agents using Algorithm 4.
2. Reduce the number of pairs using Algorithm 5.
3. Find the location of the black hole using Algorithm 1.  $\langle \rangle$

**Theorem 8.** *In oriented rings, the black hole can be found by  $k \geq 2$  dispersed anonymous agents in  $O(n \log n)$  time and cost.*

Thus, by Theorems 6 and 8, it follows that

**Corollary 5.** *Algorithm 6 is cost-optimal.*

### 4.3 Oriented Rings: Considerations on Time

In the previous section we have shown that the lower bound on the cost is tight, and can be achieved by two agents. This implies that the presence of multiple agents does not reduce the cost of locating the black hole. The natural question is whether the presence of more agents can be successfully exploited to *reduce the time complexity*.

Unlike the case of co-located agents, now the agents have to find each other to be able to distribute the workload. Note that, if the agents are able to quickly *gather* in a node, Algorithm 3 can be applied. As a consequence, in the remainder of this section, we focus on the problem to quickly group the agents.

If the number of agents  $k$  is known, the gathering problem can be easily solved by the following algorithm:

**Algorithm 7 (Gathering –  $k$  known)**

1. Each agent travels to the right using cautious walk.
2. When arriving at a node already visited by another agent, it proceeds to the right via the safe port.
3. If there is no safe port, it tests how many agents are at this node; if the number of agents at the node is  $k - 1$ , the algorithm terminates.  $\blacktriangleleft$

Eventually, since all agents travel to the right, all but one agent (which will reach the black hole) will be at the same node (in the worst case, the left neighbor of the black hole). Since, using cautious walk, it takes at most 3 time units to safely move to the right, and since there are at most  $n - 2$  such possible moves, this yields the following lemma:

**Lemma 5.** *If the numbers of agents  $k$  is known,  $k - 1$  agents can gather in an oriented ring in time  $3n - 6$ .*

This strategy can not be applied when  $k$  is unknown. In fact, while the agents can follow the same algorithm as in the previous case, they have no means to know when to terminate (and, thus, to switch to Algorithm 3).

Actually, if *causal* time complexity is considered (i.e., length of the longest chain of causally related moves, over all possible executions of the algorithm), the additional agents can be of little help in the worst case:

**Lemma 6.** *The causal time complexity of locating the black hole in an oriented ring, using  $k$  agents is at least  $n(\log n - \log k) - O(n)$ .*

However, if the *bounded delay* time complexity is considered (i.e., assuming a global clock and that each move takes at most one time unit), the additional agents can indeed help. Initially, all agents are in state *alone*.

**Algorithm 8 (Gathering –  $k$  unknown)****Rules for alone agent  $r$ :**

1. Cautiously walk to the right until you meet another agent  $r'$ .
2. If  $r'$  is in state *alone*, form a group  $\mathcal{G}$  ( $r$  and  $r'$  change state to *grouped*) and start executing the group algorithm.
3. Otherwise ( $r'$  is in state *grouped*, belonging to the group  $\mathcal{G}'$ , formed at the node  $g'$ ) join the group  $\mathcal{G}'$ : Go to  $g'$  and set your state to *Join* $[g']$ .

**Rules for group  $\mathcal{G}$  formed at node  $g$ , consisting of  $|\mathcal{G}|$  agents:**

Execute Algorithm 3 using  $|\mathcal{G}|$  agents, with the following actions taken after finishing each phase and before starting the next one:

1. If any of your agents have seen agents of another group  $\mathcal{G}'$  whose starting node  $g'$  is to the right of  $g$ , join group  $\mathcal{G}'$  by sending all your agents to  $g'$ , with state *Join* $[g']$ .
2. Otherwise add all the agents waiting at  $g$  with state *Join* $[g]$  to  $\mathcal{G}$  and execute the next phase of Algorithm 3 using the updated number of agents.  $\blacktriangleleft$

**Theorem 9.** *In oriented rings the black hole can be located by  $k = n^{1/q}$  agents in bounded delay time complexity  $O(qn^{1/q})$ .*

#### 4.4 Unoriented Ring

If the ring is unoriented, at least three dispersed agents are needed to locate the black hole (Fact 4). Thus we assume that there are at least three dispersed agents.

It is easy to convert a solution for oriented rings into one for the unoriented ones, at the cost of twice the number of moves and of agents.

**Lemma 7.** *Let  $\mathcal{A}$  be an algorithm for oriented ring which, using  $p$  agents solves problem  $\mathcal{P}$  in time  $T$  and cost  $C$ . Then there is an algorithm  $\mathcal{A}'$  for unoriented ring which, using  $2p - 1$  agents, solves  $\mathcal{P}$  in time  $T$  and complexity at most  $2C$ .*

Note that lemma 7 can be applied to all previous algorithms presented for scattered agents, except Algorithm 7.

From Theorem 8, Lemma 7 and Fact 4, it follows that:

**Theorem 10.** *Three (anonymous dispersed) agents are necessary and sufficient to locate the black hole in an unoriented ring.*

## References

1. L. Barrière, P. Flocchini, P. Fraigniaud, and N. Santoro. Gathering of mobile agents with incomparable labels. Tech. Rep. ECHO-01, Center for e-Computing, 2001.
2. M.A. Bender and D. Slonim. The power of team exploration: Two robots can learn unlabeled directed graphs. In *FOCS '94*, pages 75–85, 1994.
3. B. Brewington, R. Gray, and K. Moizumi. Mobile agents in distributed information retrieval. *Intelligent Information Agents*, pages 355–395, 1999.
4. W. Chen, C. Leng, and Y. Lien. A novel mobile agent search algorithm. *Information Sciences*, 122(2-4):227–240, 2000.
5. K. Diks, A. Malinowski, and A. Pelc. Reliable token dispersal with random faults. *Parallel Processing Letters*, 4:417–427, 1994.
6. K. Diks, A. Malinowski, and A. Pelc. Token transfer in a faulty network. *Theoretical Informatics and Applications*, 29:383–400, 1995.
7. K. Diks and A. Pelc. Fault-tolerant linear broadcasting. *Nordic Journal of Computing*, 3:188–201, 1996.
8. S. Dobrev, P. Flocchini, G. Prencipe, and N. Santoro. Finding a black hole in an arbitrary network. Tech.Rep. ECHO-03, Center for e-Computing, 2001.
9. O. Goldreich and L. Shlira. Electing a leader in a ring with link failures. *Acta Informatica*, 24:79–91, 1987.
10. O. Goldreich and L. Shlira. On the complexity of computation in the presence of link failures: The case of a ring. *Distr. Computing*, 5:121–131, 1991.
11. J. Hammer and J. Fiedler. Using mobile crawlers to search the web efficiently. *International Journal of Computer and Information Systems*. To appear.
12. F. Hohl. A framework to protect mobile agents by using reference states. In *ICDCS 2000*, 2000.
13. N. R. Jennings. On agent-based software engineering. *Art. Intelligence*, 117(2):277–296, 2000.

14. L.M. Kirousis, E. Kranakis, D. Krizanc, and Y.C. Stamatiou. Locating information with uncertainty in fully interconnected networks. In *DISC '00*, pages 283–296, 2000.
15. K. Moizumi and G. Cybenko. The travelling agent problem. *Mathematics of Control, Signal and Systems*, 1998.
16. D. Rus, R. Gray, and D. Kotz. Autonomous and adaptive agents that gather information. In *AAAI '96*, pages 107–116, 1996.
17. T. Sander and C. F. Tschudin. Protecting mobile agents against malicious hosts. In *Mobile Agents and Security*, LNCS 1419, pages 44–60, 1998.
18. K. Schelderup and J. Ines. Mobile agent security - issues and directions. In *6th Int. Conf. on Intell. and Services in Networks*, LNCS 1597, pages 155–167, 1999.

# Randomised Mutual Search for $k > 2$ Agents<sup>\*</sup>

Jaap-Henk Hoepman

Department of Computer Science, University of Twente  
P.O.Box 217, 7500 AE Enschede, the Netherlands  
`hoepman@cs.utwente.nl`

**Abstract.** We study the efficiency of randomised solutions to the mutual search problem of finding  $k$  agents distributed over  $n$  nodes. For a restricted class of so-called *linear* randomised mutual search algorithms we derive a lower bound of  $\frac{k-1}{k+1}(n+1)$  expected calls in the worst case. A randomised algorithm in the shared-coins model matching this bound is also presented. Finally we show that in general more adaptive randomised mutual algorithms perform better than the lower bound for the restricted case, even when given only private coins. A lower bound for this case is also derived.

## 1 Introduction

Buhrman *et al.* [BFG<sup>+</sup>99] introduce the mutual search problem, where  $k$  agents distributed over a complete network of  $n$  distinct nodes are required to learn each other's location. Agents can do so by calling a single node at a time to determine whether it is occupied by another agent. The object is to find all other agents in as few calls as possible. Buhrman *et al.* study this problem extensively for synchronous and asynchronous networks, and both in the deterministic and randomised case, predominantly for  $k = 2$  agents.

The prime motivation for studying this problem is the cost of conspiracy start-up in secure multi-party computations, or Byzantine agreement problems. Traditionally, this area of research assumes that all adversaries have complete knowledge of who and where they are, and that the adversaries can immediately collude to break the algorithm. The question is how hard it is to achieve this coordination, and how much information the good nodes in the system learn about the location of the adversaries during this coordination phase. For details and more examples we refer to Buhrman *et al.* [BFG<sup>+</sup>99].

Lotker and Patt-Shamir [LPS99] focus on randomised solutions to the mutual search problem in the special case of  $k = 2$  agents. They prove a lower bound of  $\frac{n+1}{3}$  expected calls in the worst case for any synchronous randomised algorithm for mutual search, and present an algorithm achieving this bound in the shared coins model.

---

<sup>\*</sup> Id: rnd-mutsearch.tex, v 1.15 2001/06/25 13:27:01 hoepman Exp

This paper is the first to consider the general case of  $k < n$  agents for randomised solutions. We generalise the results of Lotker and Patt-Shamir to a lower bound of [1]

$$\frac{k-1}{k+1}(n+1) = k-1 + \frac{k-1}{k+1}(n-k)$$

expected calls in the worst case for a restricted class of *linear* synchronous randomised algorithms (see Section 2 for an explanation of this term) that only depend on the calling history in a limited fashion. We note here that all mutual search algorithms for  $k = 2$  agents are linear by definition. We also present an algorithm achieving this bound, in the shared coins model.

Compared to the upper bound of Buhrman *et al.* [BFG<sup>+</sup>99] of

$$\frac{k(k-1)}{k(k-1)+1} \times n$$

for the deterministic case, we see that our randomised algorithm can handle roughly the square of the number of agents at the same or less cost.

Moreover, we show that more adaptive randomised mutual search algorithms outperform linear algorithms, even when the nodes are given access to a private coin only. Using shared coins we obtain a randomised algorithm whose worst case expected cost equals

$$k-1 + \left( \frac{k-1}{k+1} - \frac{k-2}{n} \right) (n-k) .$$

For the private coins model, and in the particular case where  $k = n-1$ , we present a randomised algorithms whose worst case expected cost equals  $k-1 + \frac{1}{2}$ .

Finally, we derive a lower bound of

$$k-1 + \frac{n-k}{k+1}$$

expected calls in the worst case for any randomised mutual search algorithm.

For  $k > 2$ , there are basically two choices on how to proceed when two agents find each other. They either just record each other's location and proceed to find all other agents independently, or they 'merge' into a single node where a single master agent in the merger is responsible for making all calls. In the second case, a call reaching any agent in the merged set discovers all agents in the set. Our results hold in the latter, merging agents, model. Lower bounds in this model also hold for the independent agents model.

The paper is organised as follows. In Section 2 we extend the notion of a mutual search algorithm to  $k > 2$  nodes. We then present our results on linear algorithms in Section 3. This includes the upper bound in Section 3.1, a

<sup>1</sup> The second formulation of the bound visually separates the minimal number of successful calls  $k-1$  from the fraction of unsuccessful calls among the  $n-k$  unoccupied nodes.

discussion on linear algorithms in Section 3.2, and the lower bound in Section 3.3. Next, we present the upper bounds for unrestricted mutual search algorithms in Section 4, both in the shared coins model (Section 4.1) and the private coins model (Section 4.2). Section 4.3 contains the lower bound for the unrestricted case. We conclude with some directions for further research.

## 2 Preliminaries

Let  $V$  be a set of  $n$  labelled nodes, some of which are occupied by an agent. The set of nodes occupied by an agent is called an *instance* of the mutual search problem, and is represented by a subset  $I$  of the nodes  $V$ . We use  $k = |I|$  for the number of agents. Agents are anonymous, and are only identifiable by the node on which they reside.

The nodes  $V$  lie on a completely connected graph, such that each agent can call every other node to determine whether it is occupied by another agent. If so, such a call is *successful*, otherwise it is *unsuccessful*. Nodes not occupied by agents are *passive*: they cannot make calls, and cannot store information to pass from one calling agent to the next.

Initially each agent only knows the size and labelling of the graph, the total number of agents on the graph, and the label of the node it occupies. The mutual search problem requires all agents on the graph to learn each other's location, using as few calls as possible (counting both successful and unsuccessful calls). A mutual search algorithm is a procedure that tells each agent which nodes to call, such that all agents are guaranteed to find each other. We only consider synchronous algorithms where in each time slot  $t$  a single agent makes a call. For each agent, the decision whether to make a call, and if so, which node to call, depends on the time slot  $t$ , the label of the node on which it resides, and on the result of the previous calls made and received.

With each successful call, agents are assumed to exchange everything they know so far about the distribution of agents over the graph. We distinguish two models. In the *independent agents* model, each agent is responsible to learn the location of all other agents on its own (of course using all information exchanged with a successful call). In the *merging agents* model, a successful call transfers the responsibility to find the remaining agents to the agent called (or the agent responsible for making calls on this agent's behalf<sup>2</sup>). In this model the mutual search algorithm conceptually starts with  $k$  *clans* (or equivalence classes [BFG<sup>+</sup>99]) each containing a single agent. Each successful call merges two clans into a single clan with a single *leader*, until a single clan with all  $k$  agents remains. Information flow within a clan is for free: all agents in a clan implicitly learn the location of newfound agents. As the latter model is potentially more efficient, we only consider the merging agents model.

We note that there is a trivial lowerbound of  $k - 1$  calls for any mutual search algorithm for  $k$  agents over  $n$  nodes, provided that  $k \geq 2$  and  $k < n$ . If

<sup>2</sup> This is similar to the merging fragments approach for constructing minimum weight spanning trees used by Gallager *et al.* [GHS83].



the algorithm makes less than  $k - 1$  calls, at least one of the agents did not make or receive a call and therefore cannot know the location of the other agents. For  $n = k$  (or  $k = 1$ ) the cost drops to 0, because agents know  $n$  and  $k$ .

## 2.1 Algorithm Representations

For  $k = 2$ , a deterministic synchronous mutual search algorithm can be represented by a list of directed edges  $E$ . Edge  $\overrightarrow{vw}$  is at position  $t$  in the list if an agent on node  $v$  needs to query node  $w$  at time slot  $t$ . The order in the list is represented by  $\prec$ . We will call this list the *call list*. The actual calls made will depend on the distribution of the agents over the nodes. In this setting,  $A = (V, E, \prec)$  completely describes a synchronous deterministic mutual search algorithm for  $k = 2$  agents. In fact, because the algorithm needs to make sure that any pair of agents can find each other, the graph  $(V, E)$  is a tournament.

For  $k > 2$  the picture is much less clear. As an agent learns the location of some of the other agents, it may adapt itself by changing the order of future calls, or dropping certain calls altogether. One way of describing such a mutual search algorithm would be to divide each run of the algorithm on a particular instance  $I$  in rounds. Whenever the algorithm makes a successful call (finding another agent or clan and merging the clans), a new round starts. The algorithm starts in round 1 to find the first pair of agents, and stops after the  $k - 1$ -th round when all  $k$  agents are merged into one<sup>3</sup>. Round boundaries represent knowledge changes. At such a boundary, the algorithm has to decide on a new strategy to find the next agent, and has to stick to this strategy until it finds it (or gets found). This strategy change cannot be global: only merged nodes can change their strategy, because they are the only nodes that are aware of the merge. Also, this strategy change can only depend on the agents found so far.

For a restricted class of so called linear algorithms we will derive matching upper and lower bounds. The class of linear mutual search algorithms defined below includes algorithms which are only marginally adaptive, but also *forward calling* algorithms that merge agents one by one into one single growing clan.

**Definition 2.1.** *A mutual search algorithm  $A$  to find  $k$  agents is linear, if for all instances  $I$  of size  $k + 1$*

- *$A$  makes at least  $k - 1$  successful calls<sup>4</sup> when started on instance  $I$ , and*
- *for the first  $k - 1$  successful calls  $\overrightarrow{vw}$  made by  $A$  when started on instance  $I$ ,  $A$  still makes a (now unsuccessful) call to  $w$  when started on instance  $I - \{w\}$ .*

This topic is discussed in Sect. 3.2. We note that any mutual search algorithm for  $k = 2$  agents is always linear.

<sup>3</sup> We assume that no algorithm makes calls between agents in the same clan (that already know each other's location). Therefore the graph of successful calls is acyclic, and spans all  $k$  agents when  $k - 1$  successful calls have been made.

<sup>4</sup> It is not immediately obvious that an algorithm to find  $k$  agents when started on an instance of  $k' > k$  agents will always run until it has made  $k - 1$  successful calls.

## 2.2 Properties of Binomials

The following properties of binomials are used in some proofs in this paper.

**Property 2.2.** *We have*

$$c \binom{c-1}{k-2} = (k-1) \binom{c}{k-1}. \quad (1)$$

From Feller [Fel57], p. 64, we get

$$\sum_{v=0}^r \binom{v+k-1}{k-1} = \binom{r+k}{k}. \quad (2)$$

Using this equation we derive

$$\sum_{c=k-1}^{n-1} \binom{c}{k-1} = \binom{n}{k}. \quad (3)$$

Combining Equation (II) and (3) we conclude

$$\sum_{c=k-1}^{n-1} c \binom{c-1}{k-2} = (k-1) \binom{n}{k}. \quad (4)$$

## 3 Bounds on Linear Algorithms

In this section we prove a lower bound of

$$\frac{k-1}{k+1}(n+1)$$

on the worst case expected number of calls made by any randomised mutual search algorithm to find  $k$  agents among  $n > k$  nodes, and match this with a randomised algorithm (in the shared coins model) achieving this bound. We present the algorithm first, and then prove the lower bound.

### 3.1 Upper Bound

The upper bound is proven using the following straightforward algorithm.

**Algorithm 3.1.** *First all nodes relabel the nodes according to a shared random permutation of  $\{0, \dots, n-1\}$ . The algorithm then proceeds in synchronous pulses (starting with pulse 0). An agent on node  $i$  (after relabelling) is quiet upto pulse  $i$ .*

- *If it does not receive a call at pulse  $i-1$  (or if  $i=0$ ), in pulses  $i, i+1, \dots$  it calls nodes  $i+1, i+2, \dots$  until it has contacted all  $k-1$  other agents.*
- *If a node  $i$  receives a call at pulse  $i-1$ , it remains quiet for all future pulses.*

Using this algorithm the agent on the first node (after relabelling) of the instance will call all other nodes. All other agents will remain silent. We note that the algorithm is linear.

**Theorem 3.2.** *Algorithm 3.7 has worst case expected cost*

$$\frac{k-1}{k+1}(n+1) .$$

*Proof.* Let  $I$  be a distribution of  $k$  agents over the  $n$  nodes. By performing the relabelling, the algorithm  $R$  in effect performs input-randomisation and selects a random instance  $I'$  after which it continues deterministically performing  $c(I')$  calls. Note that in pulse  $i$ ,  $R$  calls  $i+1$  if and only if there is an agent at some  $j < i+1$  and not all agents have been found yet. We conclude that  $c(I')$  equals the distance between the first and last node in  $I'$ .

The expected cost of any instance  $I$  is given by

$$\sum_{|I'|=k} c(I') \Pr[\mathcal{I} = I'] .$$

Splitting into instances with equal cost, this is equal to

$$\binom{n}{k}^{-1} \sum_c c \times \text{the number of instances } I' \text{ with } |I'| = k \text{ and cost } c . \quad (5)$$

The minimal cost is  $k-1$ , the maximal cost is  $n-1$ . For a given cost  $c$ , the range  $0, \dots, n-c-1$  of  $n-c$  nodes is a viable location for the first agent in the instance. Let  $f$  denote the position of the first agent. The last agent then resides at  $f+c$  (or else the cost would not equal  $c$ ). The remaining  $k-2$  agents can be distributed over the range  $f+1, \dots, f+c-1$  of  $c-1$  nodes. This shows that Equation (5) equals

$$\binom{n}{k}^{-1} \sum_{c=k-1}^{n-1} c(n-c) \binom{c-1}{k-2}$$

which equals

$$\begin{aligned} & \binom{n}{k}^{-1} \left( \sum_{c=k-1}^{n-1} cn \binom{c-1}{k-2} - \sum_{c=k-1}^{n-1} c^2 \binom{c-1}{k-2} \right) \\ & \quad \{\text{Using Equation (4) and (II).}\} \\ &= \binom{n}{k}^{-1} \left( n(k-1) \binom{n}{k} - (k-1) \sum_{c=k-1}^{n-1} c \binom{c}{k-1} \right) \\ &= (k-1) \binom{n}{k}^{-1} \left( n \binom{n}{k} - \sum_{c=k-1}^{n-1} (c+1) \binom{c}{k-1} + \sum_{c=k-1}^{n-1} \binom{c}{k-1} \right) \\ & \quad \{\text{Using Equation (II).}\} \end{aligned}$$

$$\begin{aligned}
&= (k-1) \binom{n}{k}^{-1} \left( n \binom{n}{k} - \sum_{c=k-1}^{n-1} k \binom{c+1}{k} + \sum_{c=k-1}^{n-1} \binom{c}{k-1} \right) \\
&\quad \{\text{Using Equation (3).}\} \\
&= (k-1) \binom{n}{k}^{-1} \left( n \binom{n}{k} - k \binom{n+1}{k+1} + \binom{n}{k} \right) \\
&= (k-1) \left( n - \frac{k(n+1)}{k+1} + 1 \right) \\
&= \frac{k-1}{k+1} (n+1)
\end{aligned}$$

This completes the proof.  $\square$

### 3.2 Linear Algorithms

In this section we give a characterisation of the class of linear mutual search algorithms as defined by Definition 2.1. This includes two important classes of mutual search algorithms defined next.

**Definition 3.3.** A mutual search algorithm  $A$  is non-adaptive, if for all instances  $I$  and for each successful call  $f$  between clan  $C$  and  $C'$  at the end of round  $i$  the following holds.

- The call list for round  $i+1$  of the merged clan  $C \cup C'$  equals the union of the call lists for round  $i$  of the clans  $C$  and  $C'$  restricted to the calls ordered after the successful call  $f$
- In the resulting call list, for each call the caller is replaced with the new leader of the clan  $C \cup C'$ .
- From this list all calls to nodes already called by one of the clans are removed, as well as duplicate calls to the same destination, in which case the first call (according to  $\prec$ ) is retained.

**Definition 3.4.** A mutual search algorithm  $A$  is forward-calling, if for all instances  $I$  and for all successful calls  $\vec{vw}, \vec{xy}$  made by  $A$  on instance  $I$

$$w = x \quad \text{implies} \quad \vec{vw} \prec \vec{xy}.$$

We have the following characterisation of linear mutual search algorithms.

**Lemma 3.5.** Both non-adaptive and forward calling mutual search algorithms are linear.

*Proof.* Observe that for a forward calling algorithm started on an instance  $I$ , each successful call of a clan reaches a clan containing a single member agent. Therefore, a forward calling algorithm grows a single clan that contains  $i$  agents during round  $i$ . If  $I$  contains  $k+1$  agents,  $A$  stops at round  $k$  when the clan contains  $k$  agents. So  $A$  makes  $k-1$  successful calls.

If  $A$  calls  $w$  in the course of the algorithm,  $w$  is the single member of its own clan. Hence  $w$  made no successful calls and was not contacted by another agent up till now. Hence removing the agent from  $w$  does not affect the strategy of  $A$  up to this point, and hence  $A$  will call  $w$ .

Observe that for a non adaptive algorithm, a clan  $C$  calls a node  $w$  iff there is a node  $v$  in the clan containing an agent for which  $\overrightarrow{vw}$  is on its initial call list (at the very start of the algorithm). If we remove the agent from  $v$ , all other agents in  $C$  will call only a subset of the nodes called by  $C$  originally, and will do so no sooner than  $C$  would have done.

Hence if for an instance  $I$  of size  $k + 1$  two clans  $C, C'$  merge at some time  $t$  by the call  $\overrightarrow{vw}$ , removing the agent from node  $w$  in  $C'$  will not result in any calls to  $C$  from the remaining agents in  $C'$  before time  $t$ . Hence  $C$  cannot distinguish these different instances, and will make the same calls (including the call to  $w$ ).

This also shows that any clan of size less than  $k$  that occurs while running the algorithm on an instance  $I$  of size  $k + 1$  also occurs on an instance  $I'$  of size  $k$  where an agent is removed from a node not in the clan. Hence, a non-adaptive algorithm cannot distinguish between instances of size  $k$  and  $k + 1$  until all agents have been found. This proves that a non-adaptive algorithm will make at least  $k - 1$  calls on an instance of size  $k + 1$ .  $\square$

### 3.3 Lower Bound

We now proceed to prove a lower bound on linear mutual search algorithms. In the proof, we use the following result of Yao [Yao77].

**Theorem 3.6** ([Yao77]). *Let  $t$  be the expected running time of a randomized algorithm solving problem  $P$  over all possible inputs, where the expected time is taken over the random choice made by the algorithm. Let  $t'$  be the average running time over the distribution of all inputs, minimized over all deterministic algorithm solving the same problem  $P$ . Then  $t > t'$ .*

Therefore, we first focus our attention to the behaviour of deterministic algorithms on random instances.

Fix  $n$  and  $k < n$ . Let  $A$  be a linear deterministic mutual search algorithm with  $|V| = n$  to locate  $k$  agents.

**Definition 3.7.** *Let  $I$  be an instance, and let  $v, w \in V$ . Define*

$$\Phi_A(\overrightarrow{vw}, I) = 1$$

*if and only if  $\overrightarrow{vw}$  is an unsuccessful call (i.e.  $v \in I, w \notin I$ ) made by algorithm  $A$  to locate all agents in the instance  $I$ .*

Then for the total cost  $C_A(I)$  of algorithm  $A$  on instance  $I$  we have

$$C_A(I) = k - 1 + \sum_{\substack{v \in I \\ w \in V - I}} \Phi_A(\overrightarrow{vw}, I), \quad (6)$$

where the  $k - 1$  term is contributed by all successful calls made.

**Lemma 3.8.** *Let  $J$  be an arbitrary set of  $k+1 \leq n$  nodes. Let  $A$  be linear. Then*

$$\sum_{\substack{v, w \in J \\ v \neq w}} \Phi_A(\vec{vw}, J - \{w\}) \geq k - 1. \quad (7)$$

*Proof.* Run  $A$  on instance  $J$ . By Definition 2.1,  $A$  will have made at least  $k - 1$  successful calls. Take the first  $k - 1$  successful calls. By Def. 2.1 for each of the first  $k - 1$  successful calls  $\vec{vw}$ , removing  $w$  does not affect the fact that  $v$  calls it. Hence  $\Phi_A(\vec{vw}, J - \{w\}) = 1$ . Hence the sum is at least  $k - 1$ .  $\square$

This lemma allows us to give a bound on the expected cost of a random instance for a given algorithm  $A$ .

**Lemma 3.9.** *Let  $A$  be a linear deterministic mutual search algorithm for  $k$  agents over  $n > k$  nodes. Then the expected cost  $E[C_A(\mathcal{I})]$  of a random instance  $\mathcal{I} = I$  of  $k$  agents is bounded by  $E[C_A(\mathcal{I})] \geq \frac{k-1}{k+1}(n+1)$ .*

*Proof.*

$$\begin{aligned} E[C_A(\mathcal{I})] &= \sum_{\substack{I \subset V \\ |I|=k}} C_A(I) \Pr[\mathcal{I} = I] \\ &= \sum_{\substack{I \subset V \\ |I|=k}} C_A(I) \binom{n}{k}^{-1} \\ &\quad \{\text{By Equation (6).}\} \\ &= \binom{n}{k}^{-1} \sum_{\substack{I \subset V \\ |I|=k}} \left( k - 1 + \sum_{\substack{v \in I \\ w \in V-I}} \Phi_A(\vec{vw}, I) \right) \\ &= k - 1 + \binom{n}{k}^{-1} \sum_{\substack{I \subset V \\ |I|=k}} \sum_{\substack{v \in I \\ w \in V-I}} \Phi_A(\vec{vw}, I) \\ &\quad \{\text{Rearranging sums, setting } J = I + \{w\}.\} \\ &= k - 1 + \binom{n}{k}^{-1} \sum_{\substack{J \subset V \\ |J|=k+1}} \sum_{\substack{v, w \in J \\ v \neq w}} \Phi_A(\vec{vw}, J - \{w\}) \\ &\quad \{\text{By Lemma 3.8}\} \\ &\geq k - 1 + \binom{n}{k}^{-1} \binom{n}{k+1} (k - 1) \\ &= \frac{k-1}{k+1} (n+1) \end{aligned}$$

Using Theorem 3.6 we now derive the following result.  $\square$

**Theorem 3.10.** *The maximal expected cost of any randomised linear mutual search algorithm over  $n$  nodes on some instance of size  $k$  is at least*

$$\frac{k-1}{k+1}(n+1) .$$

*Proof.* We first compute the expected cost of a randomised linear algorithm  $R$  on a random instance of size  $k$  of the mutual search problem over  $n$  nodes. Because the random coin flips used by  $R$  are independent of the choice of the instance we can first condition on the contents of the random tape. This fixes  $R$ , in effect making it a deterministic linear algorithm  $A$ . The expected cost over a random instance is now given by Lemma 3.9 to be at least

$$\mathbb{E}[C_A(\mathcal{I})] \geq \frac{k-1}{k+1}(n+1) .$$

As this is the expected cost of  $R$  on a random instance, there must be an instance for which the expected cost is at least this high.  $\square$

## 4 Unrestricted Algorithms

In this section we investigate the power of more adaptive algorithms whose call patterns depend heavily on the presence or absence of earlier successful calls. We show that these algorithms perform better than the lower bound for linear algorithms, given either a shared coin or a private coin.

### 4.1 Shared Coins

First we restrict our attention to algorithms deploying a shared random coin. This shared coin can be used to perform a global relabelling of nodes, as explained in section 3.1.

The following algorithm uses

$$k-1 + \left( \frac{k-1}{k+1} - \frac{k-2}{n} \right) (n-k)$$

expected calls in the worst case. In this algorithm, all agents call node 0, unless node 0 does not contain an agent, in which case the first agent that discovers this calls all other nodes until all remaining agents are found (similar to Algorithm 3.1).

**Algorithm 4.1.** *Globally relabel the nodes using the shared random coin. Agents call other nodes depending on the pulse number (starting with pulse 0) as follows.*

- An agent on node 0 (after relabelling) does not make any calls.
- An agent on node  $i > 0$  (after relabelling) is quiet upto pulse  $2i - 1$ .
  - If it did not receive any calls at earlier pulses, on pulse  $2i - 1$  it calls node 0.

- \* If this call is successful, it makes no more calls.
- \* Otherwise, at pulse  $2i, 2i + 1, \dots$  it calls nodes  $i + 1, i + 2, \dots$  until it has contacted all remaining  $k - 1$  agents.
- If an agent on node  $i > 0$  receives a call before pulse  $2i - 1$ , it remains quiet for all future pulses.

**Theorem 4.2.** Algorithm 4.1 is a mutual search algorithm for  $k$  agents on  $n$  nodes, making at most  $k - 1 + \left(\frac{k-1}{k+1} - \frac{k-2}{n}\right)(n - k)$  expected calls in the worst case.

*Proof.* We split the proof into two cases.

**an agent at node 0:** This case occurs with probability  $k/n$ . Here, all remaining  $k - 1$  agents call node 0, so that  $k - 1$  calls are made in total, and all agents are found.

**no agent at node 0:** This case occurs with probability  $(n - k)/n$ . The first agent at node  $i > 0$  that discovers this case in phase  $2i - 1$ , will call all remaining agents at nodes  $j > i$ . In fact, in this case the  $k$  agents run algorithm 3.1 on  $n - 1$  nodes. Hence the worst case expected cost is given by Theorem 3.2 as  $\frac{k-1}{k+1}n$ . We have to add 1 for the extra call to node 0 to arrive at the total cost in this case.

The worst case expected cost is therefore given by

$$\frac{k}{n}(k - 1) + \frac{n - k}{n} \left(1 + \frac{k - 1}{k + 1}n\right) = k - 1 + \left(\frac{k - 1}{k + 1} - \frac{k - 2}{n}\right)(n - k)$$

This completes the proof.  $\square$

Note that the bound of Sect. 3 can be rewritten to  $k - 1 + \frac{k-1}{k+1}(n - k)$ . Algorithm 4.1 improves this bound by  $\frac{k-2}{n}(n - k)$ .

## 4.2 Private Coins

Using private coins the agents have only limited possibilities to counter bad node assignments made by the adversary. Global relabelling is not possible, for instance, because the agents cannot exchange the outcome of their private coin tosses.

We consider the special case of  $k = n - 1$ , where all nodes except one are occupied by an agent. Here, the following algorithm uses

$$k - 1 + \frac{1}{2}$$

expected calls in the worst case.

**Algorithm 4.3.** This algorithm runs as follows.



- round 1:** If there is an agent at node 0, it either calls node 1 or node 2, each with probability  $1/2$ .
- round 2:** If the previous call was unsuccessful, the other node is called.
- round 3:** If there is an agent at node 2, which did not receive a call from node 0, it calls node 1.
- round 4:** If there is an agent at node 1, which did not receive any calls (either from node 0 or node 2), it calls node 2.
- round  $2 + i$ ,  $3 \leq i \leq n - 1$ :** If there is an agent at node 1, and node 1 did not receive a call from node 0 or was called by node 2 it calls node  $i$  at round  $i$ .
- round  $n - 1 + j$ ,  $3 \leq j \leq n - 1$ :** If there is an agent at node  $j$ , and this node did not receive a call from node 1, then it calls node 0.

**Theorem 4.4.** Algorithm 4.3 is a mutual search algorithm for  $k = n - 1$  agents on  $n$  nodes, making  $k - 1 + \frac{1}{2}$  expected calls in the worst case.

*Proof.* By case analysis, using the fact that all but one node contain an agent.

- no agent at node  $i > 2$ :** The agent at node 0 either calls node 1 (and then node 2 calls node 1 at round 3) or node 2 (and then node 1 calls node 2 at round 4). No calls are made at round  $2 + i$  for  $3 \leq i \leq n - 1$ , and all agents on nodes  $3 \leq j \leq n - 1$  call node 0 at round  $n - 1 + j$ . One of these nodes is not occupied by an agent. In this case,  $k - 1$  calls are made in total.
- no agent at node 0:** The agent at node 2 calls node 1 at round 3. Hence the agent at node 1 calls the agents at node  $i$  for  $3 \leq i \leq n - 1$  at round  $2 + i$ . Again,  $k - 1$  calls are made in total.
- no agent at node 1:** With probability  $1/2$ , the agent at node 0 calls node 2. No other calls are made in round  $2, \dots, n + 1$ . All agents on nodes  $3 \leq j \leq n - 1$  call node 0 at round  $n - 1 + j$ . Again  $k - 1$  calls are made in total. With probability  $1/2$ , the agent at node 0 calls node 1. Because this call is unsuccessful, it also calls node 2 at round 2. No other calls are made in round  $3, \dots, n + 1$ . All agents on nodes  $3 \leq j \leq n - 1$  call node 0 at round  $n - 1 + j$ . Now  $k$  calls are made in total.
- no agent at node 2:** Similar to the previous case.

The worst case occurs if there is no agent at node 1 or node 2. In either case the expected number of calls made equals  $\frac{1}{2}k + \frac{1}{2}(k - 1) = k - 1 + \frac{1}{2}$ . This completes the proof.  $\square$

### 4.3 Lower Bound

Using the results of Section 3.3 we now derive a lower bound of

$$k - 1 + \frac{n - k}{k + 1}$$

expected calls in the worst case for arbitrary randomised mutual search algorithms. We use the same notational conventions and definitions as used in that section.

**Lemma 4.5.** *Let  $J$  be an arbitrary set of  $k+1 \leq n$  nodes. Let  $A$  be an arbitrary deterministic mutual search algorithm. Then*

$$\sum_{\substack{v, w \in J \\ v \neq w}} \Phi_A(\overrightarrow{vw}, J - \{w\}) \geq 1. \quad (8)$$

*Proof.* Run  $A$  on instance  $J$ . Because  $A$  is a mutual search algorithm, it will make at least one successful call. Let  $\overrightarrow{vw}$  be the first successful call. Now remove  $w$  from  $J$ , and run  $A$  on  $J - \{w\}$ . As discussed in Section 2, the call pattern of a node depends on its initial state and all calls made so far. For  $u \neq w$ , the initial state does not change. Moreover, any calls before the call from  $v$  to  $w$  must, by assumption, be unsuccessful. Removing  $w$  does not influence the result of any of these unsuccessful call (they do not involve  $w$ ). Removing  $w$  only removes any unsuccessful calls made by  $w$  from the call pattern. They also have no effect on the call patterns of the other nodes. Hence all  $u \neq w$  make the same calls as before up to the call to  $w$ . Hence  $v$  calls  $w$ , which is now unsuccessful.  $\square$

**Theorem 4.6.** *The maximal expected cost of any randomised mutual search algorithm over  $n$  nodes on some instance of size  $k$  is at least*

$$k - 1 + \frac{n - k}{k + 1}$$

*Proof.* Using the same steps of the proof of Lemma 3.9 and using Lemma 4.5 we get

$$\begin{aligned} \mathbb{E}[C_A(\mathcal{I})] &= k - 1 + \binom{n}{k}^{-1} \sum_{\substack{J \subset V \\ |J|=k+1}} \sum_{\substack{v, w \in J \\ v \neq w}} \Phi_A(\overrightarrow{vw}, J - \{w\}) \\ &\quad \{\text{By Lemma 4.5}\} \\ &\geq k - 1 + \binom{n}{k}^{-1} \binom{n}{k+1} \\ &= k - 1 + \frac{n - k}{k + 1}. \end{aligned}$$

The theorem follows from this fact by the same reasoning as used in the proof of Theorem 3.10.  $\square$

## 5 Further Research

We have investigated the mutual search problem in the randomised case with  $k > 2$  agents. The main questions remaining are the following.

First of all, we would like to derive efficient adaptive algorithms for an arbitrary number of agents in the private coins model. Moreover, we would like to

close the gap between the lower bound and the upper bound in the shared coins model for arbitrary, non-linear, mutual search algorithms.

Secondly, the question is how changes to the model (different graphs, non-anonymous agents, independent agents, etc.) affect the cost of mutual search.

Finally, time constraints may have an adverse effect on the cost of a mutual search algorithm. If a synchronous mutual search algorithm is required to terminate within time  $t$ , with full knowledge of the distribution of the agents, the algorithm may have to make more than one call per time slot, and may not be able to fully exploit the “silence is information” paradigm.

## References

- [BFG<sup>+</sup>99] BUHRMAN, H., FRANKLIN, M., GARAY, J., HOEPFMAN, J.-H., TROMP, J., AND VITÁNYI, P. Mutual search. *J. ACM* **46**, 4 (1999), 517–536.
- [Fel57] FELLER, W. *An Introduction to Probability Theory and Its Applications*, 2nd ed. Wiley & Sons, New York, 1957.
- [GHS83] GALLAGER, R. G., HUMBLET, P. A., AND SPIRA, P. M. A distributed algorithm for minimum-weight spanning trees. *ACM Trans. Prog. Lang. & Syst.* **5**, 1 (1983), 66–77.
- [LPS99] LOTKER, Z., AND PATT-SHAMIR, B. A note on randomized mutual search. *IPL* **71**, 5–6 (1999), 187–192.
- [Yao77] YAO, A. Probabilistic computations: toward a unified measure of complexity. In *18th FOCS* (Long Beach, CA, USA, 1977), IEEE Comp. Soc. Press, pp. 222–227.

# Self-Stabilizing Minimum Spanning Tree Construction on Message-Passing Networks

Lisa Higham and Zhiying Liang

University of Calgary, Calgary, AB, T2N 1N4, Canada  
{higham,liangz}@cpsc.ucalgary.ca  
<http://www.cpsc.ucalgary.ca>

**Abstract.** Self-stabilizing algorithms for constructing a spanning tree of an arbitrary network have been studied for many models of distributed networks including those that communicate via registers (either composite or read/write atomic) and those that employ message-passing. In contrast, much less has been done for the corresponding minimum spanning tree problem. The one published self-stabilizing distributed algorithm for the minimum spanning problem that we are aware of [3] assumes a composite atomicity model. This paper presents two minimum spanning tree algorithms designed directly for deterministic, message-passing networks. The first converts an arbitrary spanning tree to a minimum one; the second is a fully self-stabilizing construction. The algorithms assume distinct identifiers and reliable fifo message passing, but do not rely on a root or synchrony. Also, processors have a safe time-out mechanism (the minimum assumption necessary for a solution to exist.) Both algorithms apply to networks that can change dynamically.

## 1 Introduction

Large networks of processors are typically susceptible to transient faults and they are frequently changing dynamically. Ideally, basic primitives used by these systems can be made robust enough to withstand these faults and adapt to network changes. Dijkstra [8] introduced a strong notion of fault tolerance called self-stabilization, that can meet these requirements. A distributed system is *self-stabilizing* if, when started from an arbitrary configuration, it is guaranteed to reach a legitimate configuration as execution progresses. If a protocol is self-stabilizing the system need not be initialized, which can be a significant additional advantage especially for physically dispersed systems such the Internet.

Two important primitives for many protocols in distributed computing are construction of a spanning tree and of a minimum spanning tree. For example, a distributed message-passing network of processors might rely on an underlying spanning tree to manage communication. If the cost of using the different communication channels varies significantly, it may be desirable to identify the spanning tree with minimum cost. So a distributed self-stabilizing (minimum) spanning tree protocol, would eventually converge to a global state where each processor has identified which of its adjacent edges are part of the required tree,

regardless of what each processor originally had identified as in the tree. Once identified, this tree would remain unchanged as long as there are no further faults and the network does not change. If further changes or faults occur, the network should again automatically adjust to identify the (possible new) tree.

Although there exist several self-stabilizing algorithms for the spanning tree problem [7,13,15,2,11,9], we are aware of only one by Antonoiu and Srimani [3] for a self-stabilizing construction of a minimum spanning tree (MST). The full version of this paper [14] contains an overview of these spanning tree papers. Perhaps the lack of minimum spanning tree solutions is because a generalization of the well-known Distributed Minimum Spanning Tree algorithm (GHS) of Gallager, Humblet and Spira [11] to the self-stabilizing setting is not apparent. The GHS algorithm resembles a distributed version of Kruskal's algorithm. It maintains a spanning forest, the components of which are merged in a controlled way via minimum outgoing edges until there is only one component, which is the MST of the network. The algorithm relies heavily on the invariant that selected edges are cycle-free, which we do not see how to maintain in the self-stabilizing setting when specific edges (of small weight) must be selected.

Antonoiu and Srimani's paper proposed the first distributed self-stabilizing algorithm for minimum spanning tree. Their algorithm is for the shared memory model and relies on composite atomicity. One approach would be to transform their algorithm to a read/write atomic solution for the link-register model via an efficient transformer (for example [6,4]) and then to apply a second transformation from the atomic read/write model to the message-passing model via self-stabilizing algorithms for token-passing and for the data-link problems as, for example, described by Dolev [10]. Both these transformation can be expensive in the worst case. So we are motivated to design directly for the message-passing model in hope of finding a more efficient and less complicated solution.

There are few self-stabilizing algorithms written for the message passing model. This paper presents two algorithms for distributed MST. The first, `Basic_MST`, does not stabilize to the MST of the network from any initial configuration; rather, it converts any valid spanning tree configuration to the minimum spanning tree in a self-stabilizing fashion. This algorithm can be used to maintain a minimum spanning tree in a network that has edges and nodes added dynamically. It also serves to provide some of the ideas for the second algorithm in a simpler but less general setting. Given `Basic_MST`, another approach to finding a general self-stabilizing MST algorithm might be to use the technique of fair composition [9,10] applied to a self-stabilizing algorithm for spanning tree construction and `Basic_MST`. However, we failed to see how to achieve this because of the need keep the variables manipulated by `Basic_MST` entirely disjoint from those used to construct the spanning tree. So our second algorithm is a general self-stabilizing deterministic minimum spanning tree algorithm, that uses some of the ideas of the first but is developed from scratch. Both algorithms differ markedly from the GHS kind of approach.

Section 2 specifies our model. We next motivate the ideas of our algorithm by outlining a sequential algorithm that constructs a MST given an arbitrary span-

ning tree (Section 3). While this algorithm is inefficient sequentially, it adapts well to a concurrent environment, and is the motivation for our first algorithm. Section 4 gives a brief sketch of this algorithm, Basic\_MST, and its correctness. All the details are available in the full paper [14]. Although one main idea is contained in Basic\_MST, substantial modifications and enhancements are needed to convert it to a general self-stabilizing MST algorithm. Section 5 presents these details and the final algorithm, Self\_Stabilizing\_MST, and its proof of correctness. Interestingly, significantly different proofs of convergence are needed for the two algorithms even though the second algorithm builds upon the ideas used in the first. Further comments and future work are briefly discussed in Section 6.

## 2 Model

An asynchronous distributed message-passing network of processors is modelled by a simple, weighted, connected and undirected graph where vertices represent processes, edges represent communication links between processes and weights represent some measure of the cost of communicating over the corresponding link. Each processor  $P$  has a distinct identifier, and knowledge only of the identifiers of its neighbours and for each neighbour  $Q$ , the weight of the edge  $\langle P, Q \rangle$ . Edge weights are assumed to be distinct since they can always be made so by appending to each weight the identifiers of the edge's end-points.

The self-stabilizing MST problem requires that given any initial configuration of the network, each processor is required to determine for each of its adjacent edges, whether or not it is in the minimum spanning tree of the network.

Self-stabilization is impossible for purely asynchronous message-passing systems [12]. We therefore assume that each process in the network is augmented with a time-out mechanism that satisfies a necessary safety property, namely: each process's time-out interval is guaranteed to be at least as long as the time taken by any message sent by the processor to travel a path of  $n$  edges where  $n$  is the number of processors in the network. For correctness we require that this lower bound on the time-out interval is not violated but it can be any (even very large) overestimate. The time-out interval could be provided directly or could be described as  $n$  times  $\alpha$ , when  $\alpha$  is the maximum time for any message to travel any edge. In the first case, knowledge of  $n$  is not required; in the second case this is the only place when knowledge of  $n$  is used. Of course, since our second algorithm is self-stabilizing, a violation in the safety of a time-out can, at worst, act as a fault from which the algorithm will eventually recover.

## 3 Graph Theory Preliminaries

We begin with a new sequential algorithm that finds the minimum spanning tree of a graph assuming that an arbitrary spanning tree is already known. This algorithm is less general and less efficient than well-known greedy solutions to the MST problem such as Kruskal's and Prim's algorithms [5], however, it has

some properties that are appealing in concurrent settings and network models and it adapts well to self-stabilization.

Let graph  $G = (V, E)$  denote a connected, weighted, undirected graph with  $n$  vertices and  $m$  edges. Let  $T = (V, E')$  denote an arbitrary spanning tree of  $G$ . A spanning tree is minimum if the sum of the weights of its edges is as small as possible. If all edge weights are distinct, the minimum spanning tree ( $MST(G)$ ) is unique. An edge  $e \in E'$  is called a *tree edge* and  $e \in E \setminus E'$  is a *non-tree edge*. If  $\langle v_0, v_1 \dots v_k \rangle$  is a path in  $T$ , and  $\langle v_k, v_0 \rangle$  is a non-tree edge, then the cycle  $\langle v_0, \dots, v_k, v_0 \rangle$  in  $G$  is called a *fundamental cycle* of  $T$  containing  $\langle v_k, v_0 \rangle$ . The proof of the following graph-theoretic propositions are from basic graph theory or are derived using standard techniques. Complete proofs are given in the full paper [14].

**Proposition 1.** *If  $e$  is in  $MST(G)$ , then  $e$  is not a maximum edge in any cycle of  $G$ .*

**Proposition 2.** *For any non-tree edge  $e$ , there is exactly one fundamental cycle of  $T$  containing  $e$ .*

Let  $fnd\_cyl(E', e)$  denote the unique fundamental cycle of  $T = (V, E')$  containing  $e$ . Denote by  $max(fnd\_cyl(E', e))$ , the edge with maximum weight in  $fnd\_cyl(E', e)$ . For non-tree edge  $e$ , the function  $minimize\_cycle(E', e)$  returns a new set of edges and is defined by

$$minimize\_cycle(E', e) = (E' \cup \{e\}) \setminus \{max(fnd\_cyl(E', e))\}.$$

**Proposition 3.** *For non-tree edge  $e$ ,  $minimize\_cycle(E', e)$  is an edge set of a spanning tree of  $G$ .*

**Proposition 4.**  *$E'$  is the edge set of  $MST(G)$  if and only if  $\forall e \in E \setminus E'$ ,  $E' = minimize\_cycle(E', e)$ .*

The next proposition says that once  $minimize\_cycle$  removes an edge from a spanning tree by replacing it with a lighter one, no subsequent application of  $minimize\_cycle$  can put that edge back into the spanning tree.

**Proposition 5.** *Consider a sequence of spanning trees  $T_0 = (V, E_0), T_1 = (V, E_1), T_2 = (V, E_2), \dots$ , where  $T_0$  is any spanning tree of  $G$ , and for  $i \geq 1$ ,  $E_i = minimize\_cycle(E_{i-1}, e_{i-1})$  for some edge  $e_{i-1} \in E \setminus E_{i-1}$ . Let  $e_i^* = max(fnd\_cyl(E_{i-1}, e_{i-1}))$ . Then  $\forall i \geq 1, e_i^* \notin E_j$  for any  $j \geq i$ .*

The preceding propositions combine to provide the strategy for an algorithm that converts an arbitrary spanning tree into the MST. Specifically, if  $minimize\_cycle$  is applied successively for each non-tree edge in any order for any initial spanning tree, the result is the minimum spanning tree.

**Proposition 6.** *Let  $T_0 = (V, E_0)$  be a spanning tree of  $G = (V, E)$  and  $\{e_1, e_2, \dots, e_{m-n}\} = E \setminus E_0$  (in any order). Let  $E_i = \text{minimize\_cycle}(E_{i-1}, e_i)$ , for  $i = 1$  to  $m - n$ . Then  $T_{m-n} = (V, E_{m-n}) = \text{MST}(G)$ .*

*Proof.* Proposition 5 implies that  $\forall e \in E \setminus E_{m-n}, e = \max(\text{find\_cyl}(E_{m-n}, e))$ . So  $E_{m-n} = \text{minimize\_cycle}(E_{m-n}, e), \forall e \in E \setminus E_{m-n}$ . So by Proposition 4,  $E_{m-n}$  is the edge set of  $\text{MST}(G)$ . ■

## 4 Construction of a Minimum Spanning Tree from a Spanning Tree

From Proposition 6 we see that the application of `minimize_cycle` to each of the non-tree edges results in the MST regardless of the order of application. This suggests that the `minimize_cycle` operations could proceed concurrently provided care is taken that they do not interfere with each other. This is the central idea for a distributed algorithm, called `Basic_MST` that identifies the minimum spanning tree of a network provided the network has already identified an arbitrary spanning tree.

Because of space constraints, we describe `Basic_MST` only informally here and give only the highlights of the proof of correctness. The full description and a detailed proof can be accessed online [14].

The description of algorithms `Basic_MST` and `Self_Stabilizing_MST` are simplified by temporarily changing perspective to one where we pretend that communication edges do the processing and that nodes act as message-passing channels between adjacent edges. Call the graph representing this particular network setting *altered*( $G$ ). That is, given a message-passing network of processors modelled by a graph  $G$ , we describe our algorithm for the network that is modeled by *altered*( $G$ ) where each edge has access to the identifiers of the edges incident at each of its end-points. It is not difficult (though notationally tedious!) to show how the original network,  $G$ , simulates an algorithm designed for the network *altered*( $G$ ) [14]. Section 6 contains an informal description of how this is done.

Each edge has a status in  $\{\text{chosen}, \text{unchosen}\}$  such that edges of the initial spanning tree are chosen and non-tree edges are unchosen. Algorithm `Basic_MST` maintains the invariant that chosen edges never form a cycle. The goal is that chosen edges eventually are exactly the edges of the MST. Recall that each edge-processor has a timer that, when initialized, has a value that is guaranteed to be as least as big as the time for a message to travel a path of length  $n$  beginning at that processor. Let *safetime*( $e$ ) be this value for edge  $e$ . An edge's timer is reset to its *safetime* upon receipt of a message that originated from that edge, otherwise the edge eventually times out. Upon time-out, each unchosen edge  $e = \langle x, y \rangle$  initiates a search for a heavier edge in the fundamental cycle containing  $e$ . Edge  $e$  does this by sending a search message containing  $e$ 's identity and weight to all edges adjacent to one of  $e$ 's end-points, say  $x$ . Any search message is discarded by any unchosen edge that did not initiate it. When a chosen edge, however, receives a search message at one end-point, it updates the message weight with



the greater of the current message weight and its own weight. It then propagates the search to all edges adjacent to its other end-point. Because chosen edges are cycle-free, at most one copy of the search message initiated by  $e$  can be received by  $e$  (necessarily at the end-point  $y$ ). All other copies die out when they reach leaves of the current tree of chosen edges. When  $e$  receives its own search message, it contains the weight of the maximum weight edge in the fundamental cycle traversed by this search message. If that weight is not  $e$ 's, then a heavier edge has been found. In this case,  $e$  initiates another message that attempts to remove the identified heavy edge and insert  $e$  into the collection of chosen edges. Otherwise,  $e$  waits for its time-out to repeat the search.

What can go wrong with this algorithm? Notice that if the searches and replaces proceed serially, then Basic\_MST is just an implementation of `minimize_cycle` repeated for all non-tree edges and hence is correct by Proposition 6. The full paper proves that the only problem that can arise due to concurrency is when a heavy edge, say  $\hat{e}$ , is identified by more than one search procedure. In this case the remove and insert procedures fail safely. Specifically, the first remove message received by  $\hat{e}$  will be successful, and will cause  $\hat{e}$  to send an insert message to its initiator. The subsequent remove messages that  $\hat{e}$  receives will be ignored because  $\hat{e}$  has changed status to an unchosen edge. Thus no corresponding insert message is generated; the initiating edge  $e$  will remain unchosen, will time-out waiting for the insert response and upon time-out will initiate a new search. To complete the proof, we focus on those steps of the scheduler that trigger the remove and insert procedures, which are called *major steps*. We first show that the weight of the chosen set decreases at every major step. We next show that if the chosen set is not the MST, then another major step must occur. Thus, the chosen set must be converted to correspond exactly to the edges of the minimum spanning tree.

Proposition 6 implies that there can be at most  $m - n$  successful remove-insert executions before the MST is identified for a network with  $n$  nodes and  $m$  edges. When distinct heavy chosen edges are identified for removal (by search messages originating from distinct non-tree edges) their replacement (by remove and insert messages) can proceed concurrently. In the worst case, however, we can construct a graph and a scheduler that force all replacements to proceed serially, each taking at most  $3 * (\text{Safetime})$  where  $\text{Safetime}$  is the maximum  $\text{safetime}(e)$ , for a total stabilization time of at most  $3(m - n)\text{Safetime}$ . Notice, however, that while the MST is being constructed, a spanning tree is maintained, which can be used in place of the MST while the fine tuning to the minimum weight tree proceeds. Furthermore, algorithm Basic\_MST will continue to adjust and converge to the MST even if the network dynamically adds new communication edges, or new nodes, or revises the weights of some edges. We need only preserve the invariant that the chosen edges form a spanning tree. Thus the MST will be automatically adjusted as edge weights are revised. Provided any new edge is inserted with status unchosen, and any new node is inserted so that exactly one of its new incident edges has status chosen, again Basic\_MST will automatically revise the chosen set to identify the new MST.

Algorithm Basic\_MST is correct of any initial values of the timers and any initial program counter values. For correctness we do, however, require that initially the chosen edges form a spanning tree, and there are no erroneous messages in the network. Some additional techniques are required to achieve a fully self-stabilizing algorithm.

## 5 A Self-Stabilizing Minimum Spanning Tree Algorithm

### 5.1 Informal Description and Intuition

Algorithm Basic\_MST guarantees convergence to the MST only if, in the initial configuration, the chosen edges form a spanning tree, and there are no messages in the network. Algorithm Self\_Stabilizing\_MST alters and enhances Basic\_MST so that the minimum spanning tree is constructed even when, initially, the chosen edges are disconnected or do not span the network or contain cycles and when there may be spurious messages already in the system.

We again described the algorithm for the altered graph where edges are assumed to do the processing. Recall that each edge processor,  $e$ , has a time-out mechanism and an associated  $\text{safetime}(e)$  that is at least as long as the time for any message it sends to travel any path of length at most  $n$ . Just as in algorithm Basic\_MST, in algorithm Self\_Stabilizing\_MST, when an unchosen edge  $e$  times out, it initiates a *search* message containing  $e$ 's identifier and weight, which propagates through chosen edges. As the propagation proceeds, the search message is updated so that it contains the weight of the heaviest chosen edge travelled by the search message. When  $e$  receives its own search message, it resets its timer to its  $\text{safetime}$  and, if it is heavier than the heaviest chosen edge travelled by the search message,  $e$  becomes passive until its next time-out. Otherwise, a heavier edge has been detected in a fundamental cycle containing  $e$ . If so  $e$  adds itself to the chosen set, and initiates a *remove* message destined for the heavy edge and intended to remove it from the chosen set.

The intuition for the enhancements is as follows. Suppose the chosen edges are disconnected or do not span the network (or both). Then there is at least one unchosen edge  $e$  whose end-points, say  $x$  and  $y$ , are not connected by a path of chosen edges. So a search message initiated by  $e$  out of its  $x$  end-point cannot return to  $e$ 's  $y$  end-point. This is detected by  $e$  through its time-out mechanism and the boolean variable *search-sent*, which indicates that  $e$  is waiting for the return of its search message. When this detection occurs  $e$  simply changes its status to chosen.

Suppose a collection of chosen edges form a cycle. To detect cycles of chosen edges, each search message is augmented to record the list of edges on the path it travelled. If a chosen edge receives a search message at one end-point, and the list in that search message contains a chosen edge that is a neighbour of its other end-point, then the search message travelled a cycle of chosen edges. This cycle-detection will succeed as long as there is an unchosen edge to initiate a search message that will travel that cycle. Another message type is needed for the case

when all edges are initially chosen and hence no search messages are generated. A *find-cycle* message is initiated by a chosen edge that timed-out because it did not received any search message within an interval equal to its time-out interval. In order to avoid initiating find-cycle messages prematurely (when there is a search message on its way to this chosen edge) the time-out interval for any chosen edge is set to three times its safetime. Like search messages, find-cycle messages record the list of chosen edges travelled, so a chosen edge receiving a find-cycle message can detect if it is in a cycle of chosen edges.

The chosen edge that detects a cycle (from either a search or a find-cycle message) initiates a remove message that travels the cycle to the edge with maximum weight in that cycle, and causes that edge to set its status to unchosen.

It is critical that the algorithm does not thrash between the procedure that changes edge status to chosen because the edge has evidence that the chosen set is disconnected, and the procedure that changes edge status to unchosen because the edge collected evidence of a cycle in the chosen set. The proof that this cannot happen and that the algorithm is correct, is assembled from several pieces as will be seen.

## 5.2 Algorithm Details

**processors:** An edge processor  $e$  has an edge identifier  $\langle u, v \rangle$ , where  $u$  and  $v$  are the distinct identifiers of its two end-points. Let EID denote the set of edge identifiers. Each edge processor  $e$  has a *weight* that is a positive integer, and is denoted by  $wt(e)$ .

The identifiers of the neighbouring edges of  $e$  at its  $u$  and  $v$  end-points are in stable storage and available to  $e$  as  $N(u)$  and  $N(v)$  respectively.

Each edge processor maintains three variables in unstable storage:

- A boolean *chosen\_status*, which indicates whether or not the edge processor  $e$  currently is in the *Chosen\_Set* subgraph.
- A non-negative integer *timer* in the interval  $[0, 3 * \text{safetime}(e)]$ , where  $\text{safetime}(e)$  is an upper bound on the time required for a message sent by  $e$  to travel any simple path in the network (necessarily of length at most  $n$ ).
- A boolean *search\_sent*, which indicates whether edge processor  $e$  has sent a search message that has not yet returned to  $e$ .

**messages:** *Search* messages have 3 fields (*search*, *eid*, *path*) where *eid* is a member of EID and *path* is a list of pairs where each pair is a member of EID and a weight. The second field records the unchosen edge that initiates the search, and the third field records the path of chosen edges travelled by the search message and those edges' weights. *Remove* messages (*remove*, *path*), and *Find-cycle* messages (*find-cycle*, *path*), each have two fields with the second field recording a path of chosen edges and weights.

**protocol:** Algorithm Self\_Stabilizing\_MST employs two procedures for edge  $\langle u, v \rangle$  to send a message. The procedure  $\text{send}(\text{mess}, e_{\text{neigh}})$  sends the message *mess* to the neighbouring edge processor with identifier  $e_{\text{neigh}}$ . (The send aborts

if  $e_{neigh}$  is not a neighbouring edge of  $\langle u, v \rangle$ ). The procedure *propagate*(*mess*, *v*) sends a copy of message *mess* to all edge processors in  $N(v)$ . The function *reset\_timer* causes an edge processor with *chosen\_status* false to reset its timer to its safetime and one with *chosen\_status* true to reset its timer to 3 times its safetime. Timers continue to decrement and cause a time-out when they reach zero. For  $list = (x_1, x_2, \dots, x_n)$ ,  $head(list) = x_1$  and  $tail(list) = (x_2, \dots, x_n)$ . The function *max\_weight*(*list*) returns the maximum weight of edges in the list. The symbol  $\oplus$  denotes concatenation. Comments are delimited by brace brackets  $\{\{\},\}$ .

### Algorithm Self\_Stabilizing\_MST

**Procedure for edge processor  $e = \langle u, v \rangle$  :**

Upon time-out:

1. If  $(\neg chosen\_status) \wedge (\neg search\_sent)$  Then
2.     *propagate*( (“search”,  $\langle u, v \rangle, \emptyset$ ), *v*);
3.     *search\_sent*  $\leftarrow$  true;
4.     Elseif  $(\neg chosen\_status) \wedge (search\_sent)$  Then  
{disconnected chosen edges}
5.     *chosen\_status*  $\leftarrow$  true;
6.     Elseif  $(chosen\_status)$  Then  
{no searches happening}
7.     *propagate* ( (“find\_cycle”,  $\{\langle u, v \rangle\}$ ), *v*);
8.     *reset\_timer*.

Upon receipt of (“search”, *sender*, *path*) from end-point, say *u*

9.     If  $(chosen\_status) \wedge (sender \neq \langle u, v \rangle)$  Then
10.     *reset\_timer*;
11.     If  $(\forall \langle v, z \rangle \in N(v), \langle v, z \rangle \notin path)$  Then  
{no cycle}
12.     *propagate* ( (“search”, *sender*,  $path \oplus \langle u, v \rangle$ ), *v*);
13.     Else  
{ $\exists \langle v, z \rangle \in N(v)$  s.t.  $\langle v, z \rangle \in path$ , so cycle of chosen}
14.     Let  $path = list_1 \oplus list_2$  where  $head(list_2) = \langle v, z \rangle$
15.     *send* ( (“remove”,  $list_2$ ),  $\langle v, z \rangle$ );
16.     Elseif  $(\neg chosen\_status) \wedge (sender = \langle u, v \rangle)$  Then  
{search traversed a find\_cyl}
17.     *reset\_timer*; *search\_sent*  $\leftarrow$  False;
18.     If  $(max\_weight(path) > wt(e))$  Then
19.     *chosen\_status*  $\leftarrow$  true;
20.     *send*( (“remove”, *path*),  $head(path)$ ).

Upon receipt of (“remove”, *path*)

20.     *reset\_timer*;
21.     If *path* is simple Then
22.     If  $(wt(e) \neq max\_weight(path))$  Then  
{not heaviest edge}
23.     *send*( (“remove”,  $tail(path)$ ),  $head(tail(path))$ );
24.     Else
25.     *chosen\_status*  $\leftarrow$  False;
26.     *search\_sent*  $\leftarrow$  False.

Upon receipt of (“find\_cycle”,  $path$ ) from  $end - point$ , say  $u$

27. **reset\_timer**;
28. If ( $chosen\_status$ ) Then
29.   If ( $\forall \langle v, z \rangle \in N(v), \langle v, z \rangle \notin path$ ) Then
30.     **propagate**((“find\_cycle”,  $path \oplus \langle u, v \rangle$ ),  $v$ );
31.   Else  $\{\exists \langle v, z \rangle \in N(v), \langle v, z \rangle \in path\}$
32.     Let  $path = list_1 \oplus list_2$  where  $head(list_2) = \langle v, z \rangle$
33.     **send**((“remove”,  $list_2$ ),  $\langle v, z \rangle$ )).

### 5.3 Correctness of Self\_Stabilizing\_MST

We prove that if algorithm Self\_Stabilizing\_MST is executed from any initial configuration, then eventually, those edges with  $chosen\_status = true$  will be exactly the edges of the minimum spanning tree and will subsequently not change. Consider any execution of algorithm Self\_Stabilizing\_MST proceeding in steps that are determined by a weakly fair scheduler. At step  $i$ , the processor chosen by the scheduler executes its next atomic action.

Let  $local\_state(e, i)$  be the sequence of  $chosen\_status(e)$ ,  $timer(e)$  and the collection of messages at  $e$  at step  $i$ . At any step  $i$ , the important attributes of the state of the entire system is captured by the configuration at step  $i$ , denoted  $Config(i)$ , and defined by:

$$Config(i) = (local\_state(e_1, i), local\_state(e_2, i), \dots, local\_state(e_m, i))$$

Define  $Chosen\_Set(i) = \{e \mid \text{In } Config(i), chosen\_status(e) = true\}$ . A  $c\_cycle$  is a cycle of edges each of which has  $chosen\_status(e)$  true.

We consider the behaviour of the network after the initial, spurious messages have been “worked out” of the system. Call a search message (“search”, sender, path), *genuine* if 1) it was initiated by an unchosen edge with edge identifier equal to sender, and 2) path is a non-cyclic path of edges starting at sender. Define genuine find\_cycle messages similarly. A remove message is *genuine* if it was generated in response to a genuine search or find\_cycle message. Let  $M$  be the maximum safetime for any edge in the system. Define step  $S_i$  to be the first step that occurs after time  $M * i$ .

**Lemma 1.** *By step  $S_1$  all messages are genuine.*

*Proof.* By the definition of safetime, any Search, Find\_cycle or Remove message that survives for  $M$  time must have travelled a path of length more than  $n$ . However Search and Find\_cycle message stop when a cycle or a fundamental cycle is detected, which must happen before  $n$  edges are traversed. A Remove message is discarded if its path contains repeated nodes. Otherwise it can travel at most along the edges in path, of which there are at most  $n$ . ■

**Lemma 2.** *Let  $\hat{V} \subseteq V$  such that the subgraph of  $(V, Chosen\_Set(i))$  induced by  $\hat{V}$  is connected and  $i \geq S_1$ . Then  $\forall$  step  $j > i$ , the subgraph of  $(V, Chosen\_Set(j))$  induced by  $\hat{V}$  is connected.*

*Proof.* Algorithm `Self_Stabilizing_MST` removes an edge from `Chosen_Set(i)` only upon receipt of a remove message (line 25). Since  $i \geq S_1$  such a message is genuine, and hence was created when a `c_cycle` was discovered. This message can remove only the unique edge with maximum weight in that `c_cycle`. So at most one edge can be removed from the `c_cycle`. Thus  $\hat{V}$  must remain connected by chosen edges. ■

**Lemma 3.** *For any initial  $\text{Config}(0)$ ,  $\forall$  steps  $i \geq S_3$ , the subgraph  $(V, \text{Chosen\_Set}(i))$  is connected and spans the network.*

*Proof.* Assume `Chosen_Set(i)` is disconnected or does not span  $G$  for all  $i, S_1 \leq i \leq S_3$ . Let  $(V_1, V_2)$  be any two subsets of  $V$  such that the edges on paths between  $V_1$  and  $V_2$  are unchosen for the interval from  $S_1$  to  $S_3$ . By step  $S_2$ , each of these unchosen edges will have timed-out, sent a search message and set `search-sent` to true. By step at most  $S_3$  none will have had its search message that it initiated returned to its opposite end-point, so each will have timed-out again. Because `search-sent` is true each will change its `chosen_status` to true. By Lemma 2, once  $V_1$  and  $V_2$  are connected they remain connected. ■

It is easy to check that if the network is itself a tree, then by step  $S_2$  all edges are chosen and will remain so. Thus `Self_Stabilizing_MST` is correct of any tree network. The remainder of this proof assumes that the network  $G$  is not a tree.

We introduce the *latent status* to capture any edge that has a Remove message destined for it. More precisely, define the *latent\_status* of edge  $e$  by: At step 0, for  $\forall e$ , `latent_status(e)` is false. `latent_status(e)` becomes true at step  $i$  if, at step  $i$ , an unchosen edge  $e'$  changes its `chosen_status` to true because of receipt of a search message (“search”, sender, path) where sender =  $e'$  and  $e$  is the edge with maximum weight in path. `latent_status(e)` becomes false at step  $j$  if, at step  $j$ , edge  $e$  changes its `chosen_status` to false because of receipt of a remove message (“remove”, path) where  $e$  is the maximum weight edge of path. Let  $\text{Latent\_Set}(i) = \{e \mid \text{In Config}(i), \text{latent\_status}(e) = \text{true}\}$

**Lemma 4.** *If  $G$  is not a tree, then for all steps  $i > S_8$ ,  $\text{Chosen\_Set}(i) \setminus \text{Latent\_Set}(i)$  is a proper subset of  $E$ .*

*Proof.* Since `Chosen_Set(i)` is connected for any  $i > S_3$ , by Lemma 3, the only way for an unchosen edge to become chosen is line 19, which also adds an edge to `Latent_Set`. The only way for `Latent_Set` to lose an edge is if that edge changes to unchosen.

Suppose  $\text{Chosen\_Set}(3M) = E$  and  $\text{Latent\_Set}(3M) = \emptyset$ . Since  $G$  is not a tree, there exists at least one `c_cycle`. If there is any search message in the network, it will propagate to this `c_cycle` and generate a remove message destined for the edge with maximum weight, say  $e$ , in that `c_cycle` by some step  $i \leq S_4$ . So  $e \in \text{Latent\_Set}(i)$ . Otherwise, some edge of the `c_cycle` will time-out within time  $3M$ , generate a `find_cycle` message, and detect the cycle in at most  $M$  additional time. So by some step  $j \leq S_8$ ,  $e \in \text{Latent\_Set}(j)$ .

Therefore, once  $\text{Chosen\_Set}(i) \setminus \text{Latent\_Set}(i)$  is a proper subset of  $E$  for  $i \geq S_8$ , it remains a proper subset for all  $j > i$ . ■

**Lemma 5.** *If  $G$  is not a tree, no chosen edge can time out after  $S_8$ .*

*Proof.* By Lemma 4, there is always an unchosen edge or a latent edge in the network. Once a chosen edge has reset its timer, a latent edge will become unchosen and an unchosen edge will time-out and propagate a search message, before a chosen edge can again time-out, because the chosen edge sets its timer to 3 times that required for any message to reach it. Because the chosen set is connected, the search message will reach any chosen edge and will cause it to reset its timer before timing out. ■

Let  $e_1, e_2, \dots, e_m$  be the edges of  $G$  sorted in order of increasing weight. Let  $\hat{E}$  be the subset of  $E$  consisting of edges of  $MST(G)$ . Define  $k(s)$  be the smallest integer in  $\{1, \dots, m\}$  such that,  $\forall i > k(s)$ ,  $e_i \in Chosen\_Set(s)$  if and only if  $e_i \in \hat{E}$ . Observe that  $k(s)$  is the index of the maximum weight edge such that the predicate  $(e_{k(s)} \in Chosen\_Set(s))$  differs from  $(e_{k(s)} \in \hat{E})$ . The proof proceeds by showing that after step  $S_8$ , we have the safety property that the index  $k(s)$  never increases, and the progress property that integer  $k(s)$  eventually decreases. Then we will be able to conclude that eventually  $k(s)$  must be 0, implying that the chosen set is the edge set of MST. All the remaining lemmas are implicitly intended to apply after step  $S_8$ .

**Lemma 6.** *For all  $e$  in  $\hat{E}$ , if  $e \in Chosen\_Set(s)$  then  $e \in Chosen\_Set(s')$   $\forall s' > s$ .*

*Proof.* The only way that edge status changes from chosen to unchosen is by receipt of a remove message, which can only remove the edge with maximum weight in some cycle of the network. However, by Proposition 1, no such edge can be in  $\hat{E}$ . ■

**Lemma 7.**  $\hat{E} \subseteq Chosen\_Set(s)$ .

*Proof.* By Lemma 6, for every  $e \in \hat{E} \cap e \in Chosen\_Set(s)$ ,  $e$  stays in  $Chosen\_Set(s')$  for  $s' > s$ . If  $\exists e \in \hat{E}$  and  $e \notin Chosen\_Set(s)$ ,  $e = \langle u, v \rangle$  will time out and initiate its search message at one end-point, say  $u$ . By Lemma 3,  $(V, Chosen\_Set(s))$  is connected and spans the network. So some copy of  $e$ 's search message will return to its  $v$  end-point. By Proposition 1, there must exist  $e'$  in the path travelled by the search message, with larger weight than  $e$ . So  $e$  becomes a chosen edge. ■

**Lemma 8.**  $k(s)$  is non-increasing.

*Proof.* By the definition of  $k(s)$ , if  $i > k(s)$  and  $e_i \in Chosen\_Set(s)$ , then  $e_i \in \hat{E}$ . By Lemma 6, for every  $s' > s$   $e_i \in Chosen\_Set(s')$ .

By the definition of  $k(s)$ , If  $i > k(s)$  and  $e_i \notin Chosen\_Set(s)$  then  $e_i \notin \hat{E}$ . Suppose  $\exists s' > s$ ,  $e_i \in Chosen\_Set(s')$ . Only line 5 or line 19 can change *chosen\_status* of  $e_i$  from unchosen to chosen. Line 5 is impossible after time  $S_3$  by Lemma 3. So  $e_i$  received its own search message ("search",  $e_i$ , *path*)

indicating the maximum weight in *path* is a chosen edge  $e_j$  other than  $e_i$ . Since  $\text{weight}(e_j) > \text{weight}(e_i)$ ,  $j > i$  because indexes of edges are by increasing weight. Hence  $j > k(s)$  and thus  $e_j$  is chosen implies  $e_j \in \hat{E}$ . But  $e_j$  is also a maximum edge in a cycle, contradicting Proposition [8](#). ■

**Lemma 9.** *If  $k(s) \geq 1$ , then  $\exists s' > s$  such that  $k(s') < k(s)$ .*

*Proof.* By the definition of  $k(s)$ ,  $e_{k(s)} \in \text{Chosen\_Set}(s)$  if and only if  $e_{k(s)} \notin \hat{E}$ . By Lemma [7](#),  $\hat{E} \subset \text{Chosen\_Set}(s)$ , so  $\text{chosen\_status}(e_{k(s)}) = \text{true}$  in  $\text{Config}(s)$  and  $e_{k(s)} \notin \hat{E}$ . Consider the unique  $\text{fnd\_cyl}(\hat{E}, e_{k(s)}) = (e_{k(s)}, e_{\alpha_1}, \dots, e_{\alpha_l})$  of edges in  $\hat{E}$  and the edge  $e_{k(s)}$ . Again by Lemma [7](#),  $e_{\alpha_i} \in \text{Chosen\_Set}(s)$  for each  $1 \leq i \leq l$ . So  $e_{k(s)}, e_{\alpha_1}, \dots, e_{\alpha_l}$  is a cycle of chosen edges in  $\text{Config}(s)$ . In this cycle  $e_{k(s)}$  must be heaviest, because by Proposition [8](#) no edge in  $\hat{E}$  can be the heaviest of any cycle of  $G$ . This cycle will be detected by some search message that traverses it, and  $e_{k(s)}$  will be removed from the chosen set at some subsequent step  $s'$ . Thus in  $\text{Config}(s')$ ,  $\text{chosen\_status}(e_{k(s)}) = \text{false}$ . So by Lemma [8](#),  $k(s') < k(s)$ . ■

**Lemma 10.** *If  $(V, \text{Chosen\_Set}(s))$  is a minimum spanning tree, then for all  $s' > s$ ,  $(V, \text{Chosen\_Set}(s'))$  is a minimum spanning tree.*

*Proof.* When  $\text{Chosen\_Set}(s)$  is a minimum spanning tree, there is no cycle of chosen edges and there is a path of chosen edges between every pair of vertices. Every unchosen edge is the largest edge in the cycle consisting of itself and the path of chosen edges between its end-points. So in  $\text{Self\_Stabilizing\_MST}$ , the unchosen edges keep timing out and sending search messages. Each search message returns to its initiator with the information that the unchosen initiator is the edge with maximum weight in the path traversed. So the unchosen edge is passive until it next times out. ■

Lemmas [8](#), [9](#) and [10](#) combine to show that our algorithm is correct.

**Theorem 1.** *Algorithm  $\text{Self\_Stabilizing\_MST}$  is a self-stabilizing solution for the minimum spanning tree problem on message-passing networks.*

## 6 Further Comments and Future Work

The presentations of algorithms  $\text{Basic\_MST}$  and  $\text{Self\_Stabilizing\_MST}$  were simplified by describing them from the “edge-processor” perspective where edges rather than nodes were assumed to drive the computation. To convert to the network model, we select one of the end-points of each edge to simulate the edge-processor. This selection could be done by simply choosing the end-point with the largest identifier, or the work could be spread more evenly by more carefully tuning this assignment. Notice that under the edge-processor descriptions, each edge is assumed to have access to the edge-identifiers of the edges incident



on either of its end-points. Thus, when end-point  $u$  simulates the computation of edge-processor  $\langle u, v \rangle$ ,  $u$  will need information about the edges adjacent to  $v$ . Typically, this information is not directly available to  $u$ . So each node uses a self-stabilizing local update algorithm to gather information on the topology up to a radius of two. The final algorithm is constructed from the fair composition of the local update algorithm and the end-point simulation of the edge-processor algorithm.

We have advertised the Internet as a possible application for our algorithms. This needs to be defended because, as was pointed out by an anonymous referee, the Internet is so large that at any time there is likely to be some fault in it. Thus, it is highly unlikely that following a fault there would be a fault-free interval as least as big as the stabilization time, especially given the large worst-case time to stabilization of our algorithms. However, the repairs in Basic\_MST and Self\_Stabilizing\_MST proceed in a distributed fashion and are typically quite independent. A fault in one part of a large network will only effect those parts of the spanning tree that might “see” that fault. For example, suppose a spanning tree edge fails by erroneously becoming unchosen due to a fault, or disappearing due to a dynamic network change. Then, only those spanning tree paths that use this edge are affected. In the Internet most connections follow approximately physically direct paths (routing from New York to Boston does not go via London), and due to caching and replication, a great deal of the traffic is relatively local. Thus most of the network spanning tree will continue to function without noticing the fault. Similar arguments can be made to defend the behaviour of the algorithms as adequate in the case of other faults or dynamic changes. Because there is no dependence on a root that must coordinate the revisions, the repairs to the spanning tree can proceed independently and “typically” locally. Note however, that there is no general local detection and correction claim that can be proved.

Setting the safetime for each edge could present a challenging trade-off in some cases. Before stabilization, time-outs trigger some essential repair mechanisms in situations that would otherwise be deadlocked. After stabilization, no errors are detected so a processor does nothing until a time-out causes it to restart error detection. Thus inflated safetimes slow convergence in some situations, but reduce message traffic after stabilization. As can be seen from the algorithm, the unstable configurations that trigger time-outs are quite specialized and may be highly unlikely to occur during stabilization in some applications. In this case it may be advantageous to choose rather large values. Further work including some simulation studies are necessary to determine appropriate safetimes for particular applications.

Our algorithms permit different safetime settings for each edge, which may be convenient since agreement does not need to be enforced. However, we do see how to exploit this possibility for efficiency while strictly maintaining self-stabilization. If faults caused by premature time-outs are tolerable, it may be reasonable to set safetime for each edge so that it is only likely to be safe rather than guaranteed. In this case it may be useful to exploit the possibility

of different time-out intervals for different edges, by setting them to reflect the time required for a message to travel the edge's fundamental cycle instead of any simple path in the network.

Both algorithms create a lot of messages some of which can be very long. The algorithms will remain impractical until these problems are addressed.

We are grateful for the thoughtful comments and careful readings by anonymous referees as provided by both the DISC and the WSS program committees.

## References

1. Aggarwal, S., Kutten, S.: Time Optimal Self-Stabilizing Spanning Tree Algorithm. In FSTTCS93 Proceedings of the 13th Conference on Foundations of Software Technology and Theoretical Computer Science, Lecture Notes in Computer Science, Vol. 761. Springer-Verlag, (1993) 400–410
2. Antonoiu, G., Srimani, P.K.: A Self-Stabilizing Distributed Algorithm to Construct an Arbitrary Spanning Tree of a Connected Graph. *Computers and Mathematics with Applications* **30** (1995) 1–7
3. Antonoiu, G., Srimani, P.: Distributed Self-Stabilizing Algorithm for Minimum Spanning Tree Construction. Euro-par'97 Parallel Processing, Lecture Notes in Computer Science, Vol. 1300. Springer-Verlag, (1997) 480–487
4. Arora, A., Nesterenko, M.: Stabilization-Preserving Atomicity Refinement. Proceedings of DISC'99, Lecture Notes in Computer Science, Springer-Verlag, (1999) 254–268
5. Atallah, M.J.: *Algorithms and Theory of Computation Handbook*. CRC Press LLC (1999)
6. Beauquier, J., Datta, A.K., Gradinariu, M., Magniette, F.: Self-Stabilizing Local Mutual Exclusion and Daemon Refinement. Proceedings of DISC'00, Lecture Notes in Computer Science, Springer-Verlag, (2000) 223–237
7. Chen, N.S., Yu, H.P., Huang, S.T.: A Self-Stabilizing Algorithm for Constructing Spanning Trees. *Information Processing Letters* **39** (1991) 147–151
8. Dijkstra, E.W.: Self Stabilizing Systems in Spite of Distributed Control. *Communications of the Association of the Computing Machinery* **17** (1974) 643–644
9. Dolev, S., Israeli, A., Moran, S.: Self-Stabilization of Dynamic Systems Assuming Only Read/Write Atomicity. *Distributed Computing* **7** (1993) 3–16
10. Dolev, S.: *Self-Stabilization*. MIT Press, Cambridge (Massachusetts) London (2000)
11. Gallager, R., Humblet, P., Spira, P.: A Distributed Algorithm for Minimum Weight Spanning Trees. *ACM Trans. on Prog. Lang. and Systems* **5:1** (1983) 66–77
12. Gouda, M.G., Multari, N.: Stabilizing Communication Protocols. *IEEE Trans. on Computers* **40** (1991) 448–458
13. Huang, S.T., Chen, N.S.: A Self-Stabilizing Algorithm for Constructing Breadth-First Trees. *Information Processing Letters* **41** (1992) 109–117
14. Liang, Z.: Self-Stabilizing Minimum Spanning Trees. Technical Report UofC/TR/TR-2001-688-11, University of Calgary, Department of Computer Science (2001)
15. Sur, S., Srimani, P.K.: A Self-Stabilizing Distributed Algorithm for BFS Spanning Trees of a Symmetric Graph. *Parallel Processing Letters* **2** (1992) 171–179

# Self Stabilizing Distributed Queuing

Maurice Herlihy and Srikanta Tirthapura

Brown University, Providence RI 02912-1910, USA  
{mph,snt}@cs.brown.edu

**Abstract.** Distributed queuing is a fundamental problem in distributed computing, arising in a variety of applications. In a distributed queuing protocol, each participating process informs its predecessor of its identity, and (when appropriate) learns the identity of its successor. This paper presents a new, self-stabilizing distributed queuing protocol. This protocol adds self-stabilizing actions to the Arrow distributed queuing protocol, a simple path-reversal protocol that runs on a network spanning tree.

The protocol is structured as a layer that runs on top of any self-stabilizing spanning tree protocol. This additional layer stabilizes in constant time, establishing that self-stabilizing distributed queuing is no more difficult than self-stabilizing spanning tree maintenance. The key idea is that the global predicate defining the legality of a protocol state can be written as the conjunction of many purely local predicates, one for each edge of the spanning tree.

## 1 Introduction

In the *distributed queuing* problem, processes in a message-passing network asynchronously and concurrently place themselves in a distributed logical queue. Specifically, each participating process informs its predecessor of its identity, and (when appropriate) learns the identity of its successor.

Distributed queuing is a fundamental problem in distributed computing, arising in a variety of applications. For example, it can be used for scalable ordered multicast [10], to synchronize access for mobile objects [11], distributed mutual exclusion (by passing a token along the queue), distributed counting (by passing a counter), or distributed implementations of synchronization primitives such as swap.

The Arrow protocol [12,4] is a simple distributed queuing protocol based on path reversal on a network spanning tree. This protocol has been used to managing mobile objects (by queuing access requests) in the Aleph Toolkit [8], where it has been shown to significantly outperform conventional directory-based schemes under high contention [11]. A recent theoretical analysis [9] has shown it to be competitive with the “optimal” distributed queuing protocol under situations of high contention.

The Arrow protocol is not fault-tolerant, because it assumes that nodes and links never fail. In this paper, we explore one approach to making the Arrow protocol fault-tolerant: *self-stabilization* [5]. Informally, a system is self-stabilizing

if, starting from an arbitrary initial global state, it eventually reaches a “legal” global state, and henceforth remains in a legal state.

Self-stabilization is appealing for its simplicity. Rather than enumerate all possible failures and their effects, we address failures through a uniform mechanism. Our self-stabilizing protocol is *scalable*: each node interacts only with its immediate neighbors, without the need for global coordination.

Of course, self-stabilization is appropriate for some applications, but not others. For example, one natural application of distributed queuing is ordered multicast, in which all participating nodes receive the same set of messages in the same order. A self-stabilizing queuing protocol might omit messages or deliver them out of order in the initial, unstable phase of the protocol, but would eventually stabilize and deliver all messages in order. Our protocol is appropriate only for applications that can tolerate such transient inconsistencies.

The key idea is that the global predicate defining the legality of a protocol state can be written as the conjunction of many purely local predicates, one for each edge of the spanning tree. We show that the delay needed to self-stabilize the Arrow protocol differs from the delay needed to self-stabilize a rooted spanning tree by only a constant. Since distributed queuing is a global relation, it may seem surprising that it can be stabilized in constant additional time by purely local actions.

We note that the protocol is *locally checkable* [3] and we could use the general technique devised by [3] to correct the state locally. But this would lead to a stabilization time of the order of the diameter of the tree, whereas our scheme gives a constant stabilization time.

## 2 The Arrow Protocol

The Arrow protocol was introduced by Kerry Raymond in [12] and later used by Demmer and Herlihy in [4] to manage distributed directories. We now give a brief and informal description of the Arrow protocol. More detailed descriptions appear elsewhere [4,10,9]. The protocol runs on a fixed spanning tree  $T$  of the network graph. Each node stores an “arrow” which can point either to itself, or to any of its neighbors in  $T$ . If a node’s arrow points to itself, then that node is tentatively the last node in the queue. Otherwise, if the node’s arrow points to a neighbor, then the end of the queue currently resides in the component of the spanning tree containing that neighbor. Informally, except for the node at the end of the queue, a node knows only in which “direction” the end of the queue lies.

The protocol is based on path reversal. Initially, one node is selected to be the head of the queue, and the tree is initialized so that following the arrows from any node leads to that head. To place itself on the queue, a node  $v$  sends a  $find(v)$  message to the node indicated by its arrow, and “flips” its arrow to point to itself. When a node  $x$  whose arrow points to  $u$  receives a  $find(v)$  message from tree neighbor  $w$ , it immediately “flips” its arrow back to  $w$ . If  $u \neq x$ , then  $x$  forwards the message to  $u$ , the prior target of its arrow. If  $u = x$  ( $x$  is tentatively

the last node in queue), then it has just learned that  $v$  is its successor. (In many applications of distributed queuing,  $x$  would then send a message to  $v$ , but we do not consider that message as a part of the queuing protocol itself.)

### 3 Model

We assume all communication links are FIFO, and that message and processor delays are bounded and known in advance. In particular, a node can *time out* if it is waiting for a response. If the time out occurs, no response will be forthcoming. (Gouda and Multari [7] have shown that such a timeout assumption is necessary for self-stabilization.)

Self-stabilizing protocols can be built in a layered fashion [13]. The protocol presented here is layered on top of a self-stabilizing rooted spanning tree protocol [12,6]. In this paper, we focus only on the upper layer, assuming that our protocol runs on a *fixed* rooted spanning tree. We show how to stabilize the arrows and the find messages.

The rest of the paper is organized as follows. Section 4 lays down the formal definitions of what it means to be a legal state and what are the possible initial states. Section 5 gives the key ideas and an informal description of the protocol. The full protocol is presented in Sect. 6 and Sect. 7 contains a proof of its correctness and a discussion of stabilization time. Section 8 contains the conclusions.

### 4 Local and Global States

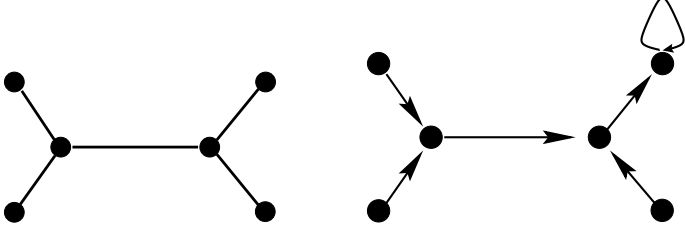
Initially, each node is in a legal local state (for example, integer variables have integer values), but local states at different nodes can be inconsistent with each other. Network edges can hold a finite number of messages. The algorithm executing at a node is fixed and incorruptible.

Recall that an underlying self-stabilizing protocol yields a rooted spanning tree  $T$  which we treat as fixed. Every node knows its neighbors in the spanning tree. As described above, in the standard Arrow protocol, each node  $v$  has a pointer denoted by  $p(v)$ . Nodes communicate by find messages.

A global state of the protocol consists of the value of  $p(v)$  for every vertex  $v$  of  $T$  (that is, the orientation of the arrows) and the set of find messages in transit on the edges of  $T$ .

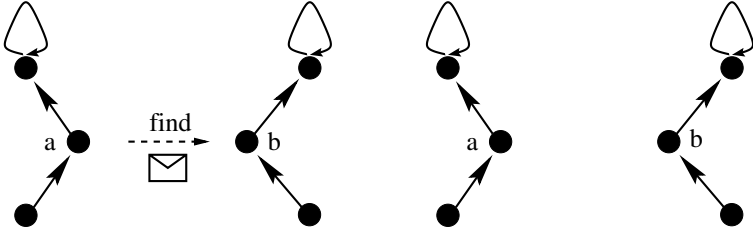
It is natural to define a legal protocol state as one that arises in a normal execution of the protocol. In the initial quiescent state, following the pointers from any node leads to a unique “sink” (a node whose arrow points to itself). A node initiates a queuing request by sending a find message to itself. When a node  $v$  gets a find message, it forwards it in the direction of  $p(v)$  and flips  $p(v)$  to point to the node where the find came from. If  $p(v)$  is  $v$ , then the find has been queued behind  $v$ ’s last request. Any of these actions is called a *find transition*. A legal execution of the protocol moves from one global state to the next via a find transition.

**Definition 1.** A state is *quiescent* if following the arrows from any node leads to a unique sink and there are no find messages in transit.



**Fig. 1.** On the left is the spanning tree  $T$ . On the right a *legal quiescent* state of the protocol

**Definition 2.** A state is *legal* either if it is a quiescent state or it can be reached from a quiescent state by a finite sequence of find transitions.



**Fig. 2.** On the left is a *legal* state which is not *quiescent*. On the right is an *illegal* state

In a possible (illegal) initial state,  $p(v)$  may point to any neighbor of  $v$  in  $T$ , and each edge may contain an arbitrary (but finite) number of find messages in transit in either or both directions. See Fig. 1 and Fig. 2

## 5 Local Stabilization Implies Global

Though the predicate defining whether a protocol state is legal or not is a global one, which depends on the values of all the pointers and the finds in transit, we show that it can be written as the conjunction of many local predicates, one for each edge of the spanning tree.

Suppose the protocol was in a quiescent state (no *finds* in transit). Let  $e$  be an edge of the spanning tree connecting nodes  $a$  and  $b$ .  $e$  divides the spanning tree into two components, one containing  $a$  and the other containing  $b$ . There is a unique sink which either lies in the component containing  $a$  or in the other component. Since all arrows should point in the direction of the sink, either  $b$  points to  $a$  or vice versa, but not both.

Now if the global state were not quiescent and there was a find message in transit from  $a$  to  $b$ , it must be true that  $a$  was pointing to  $b$  before it sent the find, but no longer is (the actions of the protocol cause the arrow turn away from the direction it just forwarded the find to the direction the find came from).  $a$  and  $b$  both point away from each other when the find is in transit.

The above cases motivate the following definition. Denote the number of find messages in transit on  $e$  by  $F(e)$ .  $p(a, e)$  is 1 if  $a$  points on  $e$  (i.e. to  $b$ ) and 0 otherwise.  $p(b, e)$  is defined similarly. For an edge  $e$ , we define  $\phi(e)$  by

$$\phi(e) = p(a, e) + p(b, e) + F(e)$$

We say that edge  $e$  is *legal* if  $\phi(e) = 1$  (either  $p(a) = b$  or  $p(b) = a$  or a find is in transit, but no two cases can occur simultaneously).

We now state and prove the main theorem of this section.

**Theorem 1.** *A protocol state is legal if and only if every edge of the spanning tree is legal.*

*Proof.* Follows from theorems 2 and 3. □

**Theorem 2.** *If a protocol state is legal, then every edge of the spanning tree is legal.*

*Proof.* In a quiescent state, there are no finds in transit and we claim that for any two adjacent nodes on the tree  $a$  and  $b$ , either  $a$  points to  $b$  or vice versa, but not both.

Clearly,  $a$  and  $b$  cannot both point to each other since we will not have a unique sink in that case. Now suppose that  $a$  and  $b$  pointed away from each other. Then we can construct a cycle in the spanning tree as follows. Suppose  $s$  was the unique sink. Following the arrows from  $a$  and  $b$  leads us to  $s$ . These arrows induce paths  $p_a$  and  $p_b$  in the tree, which intersect at  $s$  (or earlier). The cycle consists of: edge  $e$ ,  $p_a$  and  $p_b$ . Thus  $\phi(e)$  is 1 for every edge  $e$ .

Further, any find transition preserves  $\phi(e)$  for every edge  $e$ . To prove this, we observe that a find transition could be one of the following ( $v$  is a node of the tree).

- (1)  $v$  receives a find from itself; it forwards the find to  $p(v)$  and sets  $p(v) = v$
- (2)  $v$  receives a find from  $u$  and  $p(v) \neq v$ ; it forwards the find to  $p(v)$  and sets  $p(v) = u$
- (3)  $v$  receives a find from  $u$  and  $p(v) = v$ ; it queues the request at  $v$  and sets  $p(v) = u$ .

In each of the above cases, it is easy to verify that  $\phi(e)$  is preserved for every edge  $e$ . We do not do so here due to space constraints. Since every legal state is reached from a quiescent state by a finite sequence of find transitions, this concludes the proof.  $\square$

**Theorem 3.** *If every edge of the spanning tree is legal then the protocol state is legal.*

*Proof.* Let  $L$  be a protocol state where every edge is legal. Consider the directed graph  $A_L$  induced by the arrows  $p(v)$  in  $L$ . Since each vertex in  $A_L$  has out-degree 1, starting from any vertex, we can trace a unique path. This path could be non-terminating (if we have a cycle of length greater than 1) or could end at a self-loop.

**Lemma 1.** *The only directed cycles in  $A_L$  are of length one (i.e self loop).*

*Proof.* Any cycle of length greater than two would induce a cycle in the underlying spanning tree, which is impossible. A cycle of length two implies an edge  $e = (a, b)$  with  $p(a) = b$  and  $p(b) = a$ . This would cause  $\phi(e)$  to be greater than one and is also ruled out.  $\square$

The next lemma follows directly.

**Lemma 2.** *Every directed path in  $A_L$  must end in a self-loop.*

We are now ready to prove the theorem. We show that there exists some quiescent state  $Q$  and a finite sequence of find transitions  $seq$  which takes  $Q$  to  $L$ . Our proof is by induction on  $k$ , the number of find messages in transit in  $L$ .

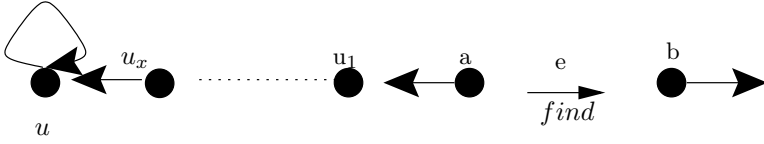
**Base case:**  $k = 0$ , i.e no find messages in transit. We prove that  $L$  has a unique sink and is a quiescent state itself and thus  $seq$  is the null sequence.

We employ proof by contradiction. Suppose  $L$  has more than one sink and  $s_1$  and  $s_2$  are two sinks such that there are no other sinks on the path connecting them on the tree  $T$ . There must be an edge  $e = (a, b)$  on this path such that neither  $p(a) = b$  nor  $p(b) = a$ . To see this, let  $n$  be the number of nodes on the path connecting  $s_1$  and  $s_2$  (excluding  $s_1$  and  $s_2$ ). The arrows on these nodes point across at most  $n$  edges. Since there are  $n + 1$  edges on this path there must be at least one edge  $e$  which does not have an arrow pointing across it. For that edge,  $\phi(e) = 0$ , making it illegal and we have a contradiction.

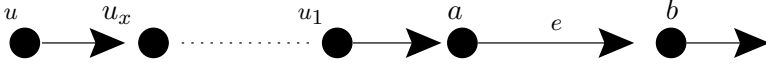
**Inductive case:** Assume that the result is true for  $k < l$ . Suppose  $L$  had  $l$  find messages in transit. Suppose a message was in transit on edge  $e$  from node  $a$  to node  $b$  (see Fig. 3). Since  $\phi(e) = 1$ ,  $a$  should point away from  $b$  and  $b$  away from  $a$ . We know from lemma 2 that the unique path starting from  $a$  in  $A_L$  must end in a self-loop. Let  $P = a, u_1 \dots u_x, u$  be that path with  $u$  having a self-loop.

Clearly, we cannot have any find messages on an edge in  $P$ , because that would cause  $\phi$  of that edge to be greater than one (an arrow pointing across the edge and a find message in transit). Consider a protocol state  $L'$  (see Fig. 4) where  $u$  did not have a self-loop. Instead,  $p(u) = u_x$  and all the arrows on path





**Fig. 3.** Global State  $L$  has a find on  $e$  and a self-loop on  $u$



**Fig. 4.** Global State  $L'$  has one find message less than  $L$ .  $L$  can be reached from  $L'$  by a sequence of find transitions

$P$  were reversed i.e.  $p(u_x) = u_{x-1}$  and so on till  $p(u_1) = a$ .  $e$  was free of find messages and  $p(a) = b$ . The state of the rest of the edges in  $L'$  is the same as in  $L$ .

We show that the  $\phi$  of every edge in  $L'$  is one. The edges on  $P$  and the edge  $e$  all have  $\phi$  equal to 1, since they have exactly one arrow pointing across them and no finds in transit. The other edges are in the same state as they were in  $L$  and thus have  $\phi$  equal to 1.

Moreover,  $L$  can be reached from  $L'$  by the following sequence of find transitions  $seq_{L',L}$ :  $u$  initiates a queuing request and the find message travels the path  $u \rightarrow u_x \rightarrow u_{x-1} \dots u_1 \rightarrow a$ , reversing the arrows on the path and is currently on edge  $e$ .

Since  $L'$  has  $l - 1$  find messages in transit and every edge of  $T$  is legal in  $L'$ , we know from induction that  $L'$  is reachable from a quiescent state  $Q$  by a sequence of find transitions  $seq_{L'}$ . Clearly, the concatenation of  $seq_{L'}$  with  $seq_{L',L}$  is a sequence of find transitions that takes quiescent state  $Q$  to  $L$ .  $\square$

## Self Stabilization on an Edge

Armed with the above theorem, our protocol simply stabilizes each edge separately. Stabilizing each edge to a legal state is enough to make the global state legal. Nodes adjacent to an edge  $e$  repeatedly check  $\phi(e)$  and “correct” it, if necessary.

The following decisions make the design and proof of the protocol simpler:

- The corrective actions to change  $\phi(e)$  are designed not to change  $\phi(f)$  for any other edge  $f$ . This is a crucial point so that now the effect of corrective actions is local to the edge only and we can prove stabilization for each edge separately.
- Out of the two adjacent nodes to an edge  $e$ , the responsibility of correcting  $\phi(e)$  rests solely with the parent node (parent in the underlying rooted spanning tree  $T$ ). The child node never changes  $\phi(e)$ .

If the value of  $\phi(e)$  could be determined locally at the parent, then we would be done. The problem though, is that  $\phi(e)$  depends on the values of variables at the two endpoints of  $e$  and on the number of find messages in transit. This can be computed by the parent after a round trip to the child and back, but the value of  $\phi(e)$  might have changed by then.

The idea in the protocol is as follows. The parent first starts an “observe” phase when it observes  $\phi(e)$ . It does not change  $\phi(e)$  during the observe phase. Since the child never changes  $\phi(e)$  anyway,  $\phi(e)$  remains unchanged when the parent is in the observe phase. The parent follows it up with a “correct” phase during which it corrects the edge if it was observed to be illegal.

The corrective actions are one of the following. We reemphasize that these change  $\phi(e)$  but don’t change  $\phi$  of any other edge of the spanning tree.  $a$  is the parent and  $b$  is the child of  $e$ .

- (1) If  $\phi(e)$  is 0, inject a new find message onto  $e$  (without any change in  $p(a)$ ), increasing  $\phi(e)$  to one.
- (2) If  $\phi(e) > 1$ , and  $p(a) = b$ , then reduce  $\phi(e)$  by changing  $p(a) = a$ .
- (3) If  $\phi(e) > 1$  but  $p(a) \neq b$ , then there must be find messages in transit on  $e$ . We show that eventually these find messages must reach  $a$  which can reduce  $\phi(e)$  by simply ignoring them.

It remains to be explained how the parent computes  $\phi(e)$ . At the start of the observe phase, it sends out an *observer* message which makes a roundtrip to the child and back. Since the edges are FIFO, by the time this returns to the parent, the parent has effectively “seen” the number of finds in transit. The observer has also observed  $p(b)$  on its way back to the parent. The parent computes  $\phi(e)$  by combining its local information with the information carried back by the observer. Once the observer returns to the parent, it enters a correct phase and the appropriate corrective action is taken.

To make the protocol self-stabilizing, we start an observe phase at the parent in response to a timeout and follow it up with a correct phase. The timeout is sufficient for two roundtrips from the parent to the child and back. If we have an observe phase followed by a “successful” correct phase, then the edge would be corrected, and would remain legal thereafter. Each observe phase has an “epoch number” to help the parent discard observers from older epochs, or maliciously introduced observers.

## 6 Protocol Description

In this section, we describe the protocol for a single edge  $e$  connecting nodes  $a$  and  $b$  where  $a$  is the parent node.

### States and Variables:

Node  $a$  has the following variables.

- (1)  $p(a)$  is  $a$ ’s pointer (or arrow), pointing to a neighbor on the tree or to itself. The rest are variables added for self-stabilization:
- (2) *state*, is boolean and is one of *observe* or *correct*.

(3) *sent* is an integer and the number of finds sent on  $e$  since the current observe phase started.

(4) *epoch* is an integer which is the epoch number of the current observe phase.

(5)  $v$  is an integer and is  $a$ 's estimate of  $\phi(e)$  when it is in a correct state.

The only variable at  $b$  is the arrow,  $p(b)$ .

**Messages:** There are two types of messages. One is the usual find message. The other is the *observer* message, which  $a$  uses to observe  $\phi(e)$ . In response to a timeout,  $a$  increments *epoch* and sends out message *observer(epoch)*, indicating the start of observe epoch *epoch*. Upon receipt,  $b$  replies with *observer(c, p(b, e))*.

**Transitions:** The transitions are of the form (event) followed by (actions). A timeout event occurs when  $a$ 's timer exceeds twice the maximum roundtrip delay from  $a$  to  $b$  and back. The timer is reset to zero after a timeout.

*Transitions for a (the parent).*

- Event: Timeout
  - Reset *state* to *observe*, *sent* to 0 and increment *epoch* {the epoch number}.
  - Send *observer(epoch)* on  $e$ .
- Event: (*state* = *observe*) and (receive find from  $b$ )
  - If  $(p(a) = a)$  then set  $p(a) \leftarrow b$  and the find is queued behind the last request from  $a$ . If  $(p(a) \neq a)$  and  $(p(a) \neq b)$ , then forward the find to  $p(a)$  and  $p(a) \leftarrow b$ . {the normal Arrow protocol actions.}
  - If  $(p(a) = b)$ , send the find back to  $b$  on  $e$ ; increment *sent*.
- Event: (*state* = *correct*) and (receive find from  $b$ ) {Eventually, *state* = *correct* implies that  $v = \phi(e)$  }
  - If  $v > 1$ , then ignore the find; decrement  $v$  {since  $\phi(e)$  has decreased}
  - Else (if  $p(a) = b$ ) send the find back to  $b$  {this situation would not arise in a legal execution}.
  - Else, normal Arrow protocol actions.
- Event: (*state* = *observe*) and (receive *observer(d, x)* on  $e$ )
  - If  $epoch \neq d$  then ignore the message. {This observer is from an older epoch or is spurious}.
  - Else change *state* to *correct*.  $v = sent + x + p(a, e)$ . {This is  $a$ 's estimate of  $\phi(e)$ , and is eventually accurate}.
  - Take corrective actions (if possible).
  - If  $v = 0$  then send find to  $a$ ; increment  $v$ .
  - If  $(v > 1$  and  $p(a) = b)$ , then  $p(a) \leftarrow a$  and decrement  $v$ .
- Event: (receive find from node  $u \neq b$  on an adjacent edge) and  $(p(a) = b)$ 
  - normal Arrow protocol actions; increment *sent* {since a find will be sent on  $e$ }.

*Actions for b (the child).*

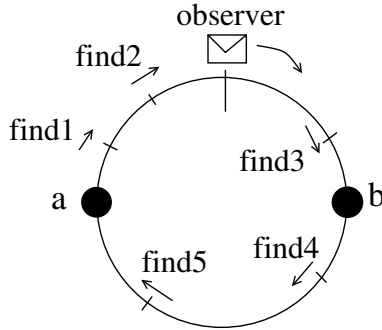
- Event: receive find from  $a$ .
  - If  $p(b) = a$ , then send find back to  $a$ .
  - Else, normal Arrow protocol actions.
- Event: receive *observer(c)*. { $a$  wants to know  $p(b, e)$ .}
  - Send *observer(c, p(b, e))* on  $e$ .

## 7 Correctness Proof

We prove two properties for every edge. The first is *closure*: if an edge enters a legitimate state then it remains in one. The second is *stabilization*: each edge eventually enters a legitimate state. We prove these properties with respect to a stronger predicate than  $\phi(e) = 1$ , since in addition to the arrows and find messages being legal, we will need to include the legality of the variables introduced for self-stabilization.

Each observer has a count (or sequence number). *sent* is a counter at *a* which is reset to zero at the beginning of every observe phase and incremented every time *a* sends out a find message on *e*. The current epoch number at *a* is *epoch*.

We visualize the edge as a directed cycle (see Fig. 5), with the link from *a* to *b* forming one half of the circumference and the link from *b* to *a* the other half. The position of the observer(s), the find messages in transit, the nodes *a* and *b* are all points on this cycle. Messages travel clockwise on this cycle and no message can “overtake” another (FIFO links). Let *R* denote the maximum roundtrip time of a message from *a* to *b* and back. *a* times out and starts a new observe phase after time  $2R$ .



**Fig. 5.** The edge is a cycle. Messages find1 and find2 belong to  $F_{al}$  and the other three finds to  $F_{la}$

Suppose there was only one “current observer”, i.e. an observer whose count matched the current epoch number (*epoch*) stored at *a*. Let *F* denote all the find messages in transit on *e*. We can divide *F* into two subsets:  $F_{al}$ , find messages between *a* and the current observer and  $F_{la}$ , the find messages between the current observer and *a*. Clearly,  $|F_{al}| + |F_{la}| + p(a, e) + p(b, e) = \phi(e)$ . If the observer is between *b* and *a*, it contains the value of  $p(b, e)$  as observed when it passed *b*. We denote this value by  $p_{obs}(b, e)$ .

**Predicate 1.** *The current observer is between a and b, and  $\phi(e) = sent + p(a, e) + p(b, e) + |F_{la}|$ .*

**Predicate 2.** *The current observer is between  $b$  and  $a$ , and  $\phi(e) = sent + p(a, e) + p_{obs}(b, e) + |F_{la}|$ .*

**Lemma 3.** *Suppose  $a$  was in an observe phase and there was only one current observer. If predicate 1 is true to start with, and  $a$  does not time out until the observer returns back, then the following will be true of the observer's trip back to  $a$ .*

- (1) *When the observer is between  $a$  and  $b$ , predicate 1 will remain true.*
- (2) *After the observer crosses  $b$  and is between  $b$  and  $a$ , predicate 2 will be true.*
- (3) *When the observer returns to  $a$ ,  $a$  enters a correct state where  $v = \phi(e)$ .*

*Proof.* Until the observer returns to  $a$ , it will remain in an observe phase, and  $\phi(e)$  will not change. Recall that  $\phi(e) = |F| + p(a, e) + p(b, e)$ .

As long as the observer hasn't reached  $b$ , every find in  $F_{al}$  must have been injected by  $a$  after the observer left  $a$  and thus  $F_{al} = sent$ . We have  $\phi(e) = p(a, e) + p(b, e) + |F_{la}| + |F_{al}| = p(a, e) + p(b, e) + |F_{la}| + sent$ , satisfying predicate 1. This proves part (1).

Suppose the observer is just about to cross  $b$ . We have  $\phi(e) = sent + p(a, e) + p(b, e) + |F_{la}|$ . Immediately after the observer crosses  $b$ , we have  $p_{obs}(b, e) = p(b, e)$ . Since  $\phi(e)$  has not changed and none of the other quantities  $p(a, e)$ ,  $|F_{la}|$  have changed in the meanwhile (think of the observer crossing  $b$  as an atomic operation)  $\phi(e) = sent + p(a, e) + p_{obs}(b, e) + |F_{la}|$  after the observer has crossed, and predicate 2 is true.

We now prove by induction over the size of  $|F_{la}|$  that predicate 2 continues to hold. Suppose it was true when  $|F_{la}|$  was  $k$ . If  $|F_{la}|$  decreases to  $k - 1$ , then a find must have been delivered to  $a$ . If  $p(a, e)$  was 1, then the find would have bounced back on  $e$  and  $sent$  would have increased by 1. If  $p(a, e)$  was zero ( $a$  was pointing away from  $b$ ), then  $p(a, e)$  would increase to 1 and  $sent$  would remain the same. In either case, the sum  $sent + p(a, e)$  would increase by 1, and  $|F_{la}| + sent + p(a, e)$  would remain unchanged. This proves part (2).

When the observer reaches  $a$  again,  $F_{la}$  will be the empty set and we thus have  $\phi(e) = sent + p(a, e) + p_{obs}(b, e)$ . Once the observer reaches  $a$ ,  $a$  will enter a correct state and sets  $v$  to the above quantity ( $sent + p(a, e) + p_{obs}(b, e)$ ). This proves part (3).  $\square$

We will now define the set of legitimate states for an edge, this time including the variables introduced for self-stabilization as well.

- $R_1$  denotes the predicate:  $\phi(e) = 1$ .
- We denote the AND of the following predicates by  $R_2$ .
  - (1)  $a$ 's state is observe
  - (2) there is exactly one current observer
  - (3) the other observers have counts less than  $epoch$  (the current epoch number at  $a$ )
  - (4) predicates 1 or 2 should be satisfied

- We denote the AND of the following predicates by  $R_3$ .
  - (1)  $a$ 's state is correct
  - (2) there is no observer with a count greater than or equal to  $epoch$
  - (3)  $v = \phi(e)$  (i.e.,  $a$  knows  $\phi(e)$ )

Since  $a$  can be in either the observe state or in the correct state, but never both at the same time, only one of  $R_1$  or  $R_2$  can be true at a time.

**Definition 3.** *The edge is in a legitimate state iff the following is true:  $R_1 \wedge (R_2 \vee R_3)$ .*

To prove self-stabilization of the protocol, we first prove stabilization and closure for the predicate  $R_2 \vee R_3$  and then for the predicate  $R_1 \wedge (R_2 \vee R_3)$ . This technique has been called a *convergence stair* in [7].

**Lemma 4.** *If  $R_2 \vee R_3$  is true, then it will continue to remain true.*

*Proof.* We consider two cases and all the possible actions that could occur in each case.

(1)  $R_3$  is true. As long as  $a$  is in the correct state, it will not introduce any new observers.  $v = \phi(e)$  to start with; any changes to  $\phi(e)$  are made at  $a$  and are also reflected in  $v$ , thus  $v$  will remain equal to  $\phi(e)$ . Suppose  $a$  times out and enters an observe state, it increments  $epoch$  and sends out an observer with sequence number  $epoch$ . There is only one current observer and it satisfies predicate 1 trivially.  $R_2$  is true now and so is  $R_2 \vee R_3$ .

(2)  $R_2$  is true. If the observer does not reach  $a$  and  $a$  does not time out, then the observer remains on the edge  $e$  and predicate 1 or predicate 2 will continue to hold due to lemma 3. If  $a$  times out before the observer reaches it, then it will enter an observe state,  $epoch$  is incremented, and a new observer is injected into the channel with sequence number  $epoch$ .  $R_2$  is still true (only one current observer; no observers with sequence number greater than  $epoch$ ; predicate 1 is true). If the observer reaches  $a$  before it times out, then  $a$  will go to a correct state and by lemma 3,  $v = \phi(e)$  at  $a$ . Thus  $R_3$  is true at  $a$ .  $\square$

**Lemma 5.** *Closure: If  $R_1 \wedge (R_2 \vee R_3)$  is true, then it will continue to remain true.*

*Proof.* From lemma 4, we know that  $R_2 \vee R_3$  will continue to remain true.

We have to show that  $R_1$  will continue to hold. Since  $R_1$  is true initially, we have  $\phi(e) = 1$  to start with. If we can show that  $\phi(e)$  is never changed, then we are done.

If  $a$  is in the observe state ( $R_2$  is true), then  $\phi(e)$  is never changed. If  $a$  is in the correct state ( $R_3$  is true), we have  $v = \phi(e) = 1$  (by predicate  $R_3$ ). If  $v = 1$ , then  $a$  will not take any corrective action, and thus  $\phi(e)$  is never changed.  $\square$

**Definition 4.** *A state is fresh if  $a$  has just timed out, and thus the next time out is  $2R$  time steps away. It is half-fresh if the next time out is at least  $R$  time steps away.*

**Lemma 6.** *Within  $3R$  time of any state, we will reach a state where  $R_2$  is true and the state is fresh.*

*Proof.* If there are any observers in transit with sequence numbers greater than  $epoch$ , then they will reach  $a$  within time  $R$  (the roundtrip time). All the observers that  $a$  injects have sequence numbers less than or equal to  $epoch$ . Clearly, after time  $R$  there will never be an observer with sequence number greater than  $epoch$ .

Within time  $2R$  after that,  $a$  will time out, enter an observe state, increment  $epoch$  and send out a new observer resetting  $r$  to zero. Clearly  $R_2$  is now true; there is only one current observer, the other observers have sequence numbers less than  $epoch$ , and predicate 1 is true. Since  $a$  has just timed out, the state is fresh.  $\square$

**Lemma 7.** *Starting from any state, within  $4R$  time we will reach a state where  $R_3$  is true and the state is half-fresh.*

*Proof.* From lemma 6 we know that  $R_2$  will be true within time  $3R$ . Thus  $a$  is in an observe state and its current observer obeys predicates 1 or 2. If the observer reaches  $a$  before  $a$  times out (going into an observe state of a later epoch), then  $a$  will enter a correct state. And by lemma 3  $v = \phi(e)$ . Thus  $R_3$  will be true at  $a$ .

Since the state we started out is fresh, the next time out will occur only after time  $2R$ . Since  $R$  is the maximum roundtrip time, the observer will indeed reach  $a$  within time  $R$  (before the timeout), and the next time out is at least  $R$  away. Thus the state is half-fresh.  $\square$

**Lemma 8.** *Stabilization: In time  $5R$  we will reach a state where  $R_1 \wedge (R_2 \vee R_3)$  is true.*

*Proof.* From lemma 7 within time  $4R$ , we are in a state where  $R_3$  (and thus  $R_2 \vee R_3$ ) is true and the state is half-fresh. From lemma 4 we know that  $R_2 \vee R_3$  will remain true after that. We now show that within  $R$  more time steps, predicate  $R_1$  will also be true.

If  $R_3$  is true then  $v = \phi(e)$ . If  $\phi(e) = 1$ , then  $R_1$  is already true. If  $\phi(e) = 0$ , then  $a$  will increase  $\phi(e)$  immediately and  $R_1$  will be true. If  $\phi(e) > 1$  and  $p(a) = b$ , then  $a$  reduces  $\phi$  by setting  $p(a) = a$ .

We are now left with the case when  $\phi(e) > 1$  and  $p(a) \neq b$ . Since  $\phi(e) = p(a, e) + p(b, e) + |F|$  (where  $F$  is the set of all find messages in transit), and  $p(a, e) = 0$ , it must be true that  $|F| > 0$ . Let  $\phi_c$  be the current value of  $\phi(e)$ . We prove that within  $R$  time steps, at least  $\phi_c - 1$  find messages must arrive at  $a$  on  $e$ . Since the timeout is at least  $R$  time away (the state is half-fresh),  $a$  remains in a correct state for at least time  $R$  and by ignoring all those find messages, it reduces  $\phi(e)$  to 1, thus satisfying  $R_1$ .

We now show that at least  $\phi_c - 1$  find messages arrive at  $a$  within the next  $R$  time steps. We use proof by contradiction. Let the current state of the system

be *start*, the state of the system after  $R$  time steps be *finish* and the state of the system after the maximum latency between  $a$  and  $b$  be *middle*. The time interval between *middle* and *finish* is more than the maximum latency for the link between  $b$  and  $a$ , since  $R$  is greater than the sum of the maximum  $a \rightarrow b$  and  $b \rightarrow a$  latencies.

Suppose less than  $\phi_c - 1$  messages arrived at  $a$  in time  $R$ . This means that all the while from *start* to (and including) *finish*,  $v = \phi(e) > 1$  and  $p(a) \neq b$ . Thus,  $a$  never forwarded any finds onto  $e$  after *start*. After *middle*, and until *finish*, there will be no finds in transit from  $a$  to  $b$ , since all the find messages that were in transit at *start* would have reached  $b$ . We have two cases.

If  $p(b, e) = 0$  in *middle*, then  $b$  will not forward any more finds on  $e$  till *finish*. By the time we reached *finish*, all finds that were in transit from  $b$  to  $a$  in *middle* would have reached  $a$ . At *finish*, there are no find messages in transit on  $e$  ( $a$  did not send any after *start* and neither did  $b$  after *middle*) and  $p(a, e) = 0$  and  $p(b, e) = 0$ , and thus  $\phi(e) = 0$  in *finish*, which is a contradiction.

If  $p(b, e) = 1$  in *middle*, then  $b$  might forward a find on  $e$  between *middle* and *finish* and this would change  $p(b, e)$  to 0. But since no finds are forthcoming from  $a$ , this is the last find that would arrive from  $b$  before *finish*. The rest of the finds would have reached  $a$  by *finish* and thus at *finish*, the value of  $\phi(e)$  is at most 1 ( $\leq 1$  finds in transit,  $p(a, e) = 0$  and  $p(b, e) = 0$ ), which is again a contradiction.  $\square$

**Stabilization time:** From lemma 8, each edge will stabilize to a legitimate state in  $5R$  time steps where  $R$  is the maximum roundtrip time for that edge. The protocol stabilizes when the last edge has stabilized. Thus the stabilization time of the protocol is  $5R_{max}$  where  $R_{max}$  is the maximum of the roundtrip times of all the edges.

## 8 Conclusions

We have presented a self-stabilizing Arrow queuing protocol. This was possible because of a decomposition of the global predicate defining “legality” of a protocol state into the conjunction of a number of purely local predicates, one for each edge of the spanning tree. The delay needed to self-stabilize the Arrow protocol differs from the delay needed to self-stabilize a rooted spanning tree by only a constant number of round trip delays on an edge.

**Acknowledgments.** The second author is grateful to Steve Reiss for helpful discussions and ideas.

## References

- [1] S. Aggarwal and S. Kutten. Time optimal self-stabilizing spanning tree algorithm. In *FSTTCS93 Proceedings of the 13th Conference on Foundations of Software Technology and Theoretical Computer Science*, Springer-Verlag LNCS:761, pages 400–410, 1993.



- [2] G. Antonoiu and P. Srimani. Distributed self-stabilizing algorithm for minimum spanning tree construction. In *Euro-par'97 Parallel Processing, Proceedings LNCS:1300*, pages 480–487. Springer-Verlag, 1997.
- [3] B. Awerbuch, B. Patt-Shamir, and G. Varghese. Self-stabilization by local checking and correction. In *FOCS91 Proceedings of the 31st Annual IEEE Symposium on Foundations of Computer Science*, pages 268–277, 1991.
- [4] M. Demmer and M. Herlihy. The arrow directory protocol. In *Proceedings of 12th International Symposium on Distributed Computing*, Sept. 1998.
- [5] E. Dijkstra. Self stabilizing systems in spite of distributed control. *Communications of the ACM*, 17:643–644, 1974.
- [6] S. Dolev, A. Israeli, and S. Moran. Self-stabilization of dynamic systems assuming only read/write atomicity. *Distributed Computing*, 7:3–16, 1993.
- [7] M. Gouda and N. Multari. Stabilizing communication protocols. *IEEE Transactions on Computers*, 40:448–458, 1991.
- [8] M. Herlihy. The aleph toolkit: Support for scalable distributed shared objects. In *Workshop on Communication, Architecture, and Applications for Network-based Parallel Computing (CANPC)*, January 1999.
- [9] M. Herlihy, S. Tirthapura, and R. Wattenhofer. Competitive concurrent distributed queuing. In *Proceedings of the ACM Symposium on Principles of Distributed Computing (to appear)*, August 2001.
- [10] M. Herlihy, S. Tirthapura, and R. Wattenhofer. Ordered multicast and distributed swap. *Operating Systems Review*, 35(1):85–96, January 2001.
- [11] M. Herlihy and M. Warres. A tale of two directories: implementing distributed shared objects in java. *Concurrency - Practice and Experience*, 12(7):555–572, 2000.
- [12] K. Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems*, 7(1):61–77, 1989.
- [13] M. Schneider. Self-stabilization. *ACM Computing Surveys*, 25:45–67, 1993.
- [14] G. Varghese. Self-stabilization by counter flushing. In *Proceedings of the Thirtieth Annual ACM Symposium on Principles of Distributed Computing*, pages 244–253, 1994.
- [15] G. Varghese, A. Arora, and M. Gouda. Self-stabilization by tree correction. *Chicago Journal of Theoretical Computer Science*, (3):1–32, 1997.

# A Space Optimal, Deterministic, Self-Stabilizing, Leader Election Algorithm for Unidirectional Rings

Faith E. Fich<sup>1</sup> and Colette Johnen<sup>2</sup>

<sup>1</sup> Department of Computer Science, University of Toronto, Canada  
fich@cs.toronto.edu

<sup>2</sup> Laboratoire de Recherche en Informatique, CNRS–Université de Paris-Sud, France  
colette@lri.fr

**Abstract.** A new, self-stabilizing algorithm for electing a leader on a unidirectional ring of prime size is presented for the composite atomicity model with a centralized daemon. Its space complexity is optimal to within a small additive constant number of bits per processor, significantly improving previous self-stabilizing algorithms for this problem. In other models or when the ring size is composite, no deterministic solutions exist, because it is impossible to break symmetry.

## 1 Introduction

Electing a leader on a ring is a well studied problem in the theory of distributed computing, with recent textbooks devoting entire chapters to it [193]. It requires exactly one processor in the ring be chosen as a leader. More formally, there is a distinguished subset of possible processor states in which a processor is considered to be a leader. The state of the processor that is chosen leader reaches and then remains within the subset, whereas the states of all other processors remain outside the subset. A related problem of interest is token circulation, where a single token moves around the ring from processor to processor, with at most one processor having the token in any configuration. The formal definition of a processor having a token is that the state of the processor belongs to a subset of distinguished states.

Self-stabilizing algorithms are those which eventually achieve a desired property (for example, having a unique leader) no matter which configuration they are started in (and, hence, after transient faults occur). Dijkstra introduced the concept of self-stabilization and gave a number of self-stabilizing algorithms for token circulation on a bidirectional ring, assuming the existence of a leader [9]. One of these algorithms uses only 3 states per processor [11]. On a unidirectional ring with a leader, Gouda and Haddix [13] can perform self-stabilizing token circulation using 8 states per processor.

Conversely, given a self-stabilizing token circulation algorithm on a ring of size  $n$ , there is an easy self-stabilizing algorithm to elect a leader using only an additional  $\lceil \log_2 n \rceil$  bits per processor. Specifically, each processor stores a name

from  $\{0, \dots, n-1\}$  as part of its state. Whenever a processor gets the token, it updates its name by adding 1 to the name of its left neighbour and then taking the result modulo  $n$ . Eventually, there will a unique processor with name 0, which is the leader.

Without the ability to break symmetry, deterministic self-stabilizing leader election and token circulation are impossible [2]. For example, consider a synchronous system of anonymous processors. If all processors start in the same state with the same environment, they will always remain in the same state as one another. Similarly, in an asynchronous shared memory system of anonymous processors with atomic reads and writes, where all registers have the same initial contents, or in an asynchronous message passing system of anonymous processors, where all communication links contain the same nonempty sequence of messages, many schedules, for example, a round robin schedule, will maintain symmetry among all the processors. Therefore, the study of deterministic algorithms for leader election and token passing in systems of anonymous processors has focussed on Dijkstra's composite atomicity model with a centralized daemon (where a step consists of a state transition by a single processor, based on its state and the states of its neighbours). Even in this model, symmetry among equally spaced, nonadjacent processors in a ring can be maintained by an adversarial scheduler. Therefore, deterministic algorithms for leader election and token circulation are possible in a ring of  $n$  anonymous processors only when  $n$  is prime [2][10][7].

Randomization is a well known technique to break symmetry and randomized algorithms for both problems have been considered on a variety of models [1][15][4]. This work is beyond the scope of our paper.

There are deterministic self-stabilizing leader election algorithms for bidirectional rings (of prime size using the composite atomicity model with a centralized daemon) that use only a constant amount of space per processor [17]. For unidirectional rings, Burns and Pachl [7] presented a deterministic self-stabilizing token circulation algorithm that uses  $O(n^2)$  states per processor, as well as a more complicated variant that uses  $O(n^2/\log n)$  states per processor. They left the determination of the space complexity of this problem as an open question. Lin and Simon [18] further improved their algorithm to  $O\left(n\sqrt{n/\log n \log \log n}\right)$  states per processor.

Beauquier, Gradinariu, and Johnen [4] proved a lower bound of  $n$  states per processor for any deterministic self-stabilizing leader election algorithm on a unidirectional ring of size  $n$ . They also mentioned a similar lower bound of  $(n-1)/2$  states per processor for token circulation due to Jaap-Henk Hoepman.

In Section 3, we present a deterministic self-stabilizing leader election algorithm for unidirectional rings (of prime size using the composite atomicity model with a centralized daemon) in which the number of states is  $O(n)$ . This matches the lower bound to within a small constant factor. Hence, our algorithm matches the number of bits of storage used at each processor to within a small additive constant of the number required by the lower bound. An algorithm for self-stabilizing token circulation on a unidirectional ring can be obtained by

combining our algorithm for electing a leader with Gouda and Haddix's token circulation algorithm that assumes the existence of a leader [13]. The resulting algorithm has provably optimal space complexity to within a small additive constant, solving Burns and Pachl's open question.

Our algorithm was inspired by and is closely related to Burns and Pachl's basic algorithm. To achieve small space, our idea is to time share the space: two pieces of information are stored alternately in one variable instead of in parallel using two different variables. However, the correct implementation of this simple idea in a self-stabilizing manner is non-trivial.

When developing our algorithm, we use Beauquier, Gradinariu, and Johnen's alternating schedule approach [45] to simplify the description and the proof of correctness. In Section 2, we give a more careful description of our model of computation, define the set of alternating schedules, and state some important properties of executions that have alternating schedules. Most of these results describe how information flows from one processor to another during the course of an execution.

## 2 The Model

We consider a system consisting of  $n$  identical, anonymous processors arranged in a ring, where  $n$  is prime. The value of  $n$  is known to the processors. The left neighbour of a processor  $P$  will be denoted  $P_L$  and its right neighbour will be denoted  $P_R$ . The ring is *unidirectional*, that is, each processor can only directly get information from its left neighbour. The *distance* from processor  $P$  to processor  $Q$  is measured starting from  $P$  and moving to the right until  $Q$  is reached. In particular, the distance from  $P$  to  $P_L$  is  $n - 1$ .

In any algorithm, each processor is in one of a finite number of states. A *configuration* specifies the state of every processor. An *action* of a processor is a state transition, where its next state depends on its current state and the state of its left neighbour. Note that the next state might be the same as the current state. Only one processor performs an action at a time. This is Dijkstra's composite atomicity model with a centralized daemon [9]. An algorithm is *deterministic* if, for each processor, its actions can be described by a total state transition function from the cross product of its state set and the state set of its left neighbour. In other words, in every configuration, each processor has exactly one action it can perform. A processor is *enabled* in a configuration if there is an action that causes the processor to change its state. Some authors prefer to describe a deterministic algorithm using partial state transition functions, not defining those transitions in which a processor is not enabled.

A *schedule* is a sequence whose elements are chosen from the set of  $n$  processors. If  $P$  is the  $t$ 'th element of the schedule, we say that  $t$  is a *step* of  $P$  and  $P$  takes a step at time  $t$ . A processor  $P$  takes a step during the time interval  $[t, t']$  if it takes a step at some time  $t'$ , where  $t \leq t' \leq t''$ . In particular, if  $t > t''$ , then the interval  $[t, t'']$  is empty and no processor takes a step during this interval.

An *execution* is an infinite sequence of configurations and processors

$$\text{config}(0), \text{proc}(1), \text{config}(1), \text{proc}(2), \text{config}(2), \dots$$

where configuration  $\text{config}(t)$  is obtained from configuration  $\text{config}(t-1)$  by the action of processor  $\text{proc}(t)$ , for all  $t > 0$ . We say that  $\text{config}(t)$  is the configuration at time  $t$  of the execution. The initial configuration of this execution is  $\text{config}(0)$ , the configuration at time 0. The *schedule of the execution* is the subsequence  $\text{proc}(1), \text{proc}(2), \dots$  of processors. An infinite schedule or execution is *fair* if every processor appears in the sequence infinitely often.

Fix an execution. If processor  $P$  takes a step at time  $T$ , then the state of  $P$  at time  $T$  is influenced by the state of  $P_L$  at time  $T-1$ . If  $P$  does not take any steps in the interval  $[T+1, t']$ , then it will have the same state at  $T$  and  $t'$ . Similarly, if  $P_L$  does not take any steps in the interval  $[t+1, T]$ , then it will have the same state at  $t$  and  $T-1$ . Thus the state of  $P_L$  at time  $t$  influences the state of  $P$  at time  $t'$ . The following definition extends this relationship to pairs of processors that are further apart.

**Definition 1.** Suppose  $P_0, P_1, \dots, P_k$  are  $k+1 \leq n$  consecutive processors, in order, rightwards along the ring. Then the state of  $P_0$  at time  $t_0$  **influences** the state of  $P_k$  at time  $t' > t_0$ , denoted

$$(P_0, t_0) \rightarrow (P_k, t'),$$

if and only if there exist times  $t_0 < t_1 < \dots < t_k \leq t'$  such that  $P_k$  takes no steps during the time interval  $[t_k+1, t']$  and, for  $i = 1, \dots, k$ ,  $P_{i-1}$  takes no steps during the time interval  $[t_{i-1}+1, t_i]$ , but  $P_i$  takes a step at time  $t_i$ .

This definition of influence only captures the communication of information around the ring. It does not capture knowledge that a processor retains when it takes steps. For example, if  $P$  takes a step at time  $T$ , then  $(P, T-1) \not\rightarrow (P, T)$ . The following results are easy consequences of the definition.

**Proposition 1.** Suppose  $P$ ,  $P'$ , and  $P''$  are distinct processors with  $P'$  on the path from  $P$  to  $P''$ . If  $(P, t) \rightarrow (P'', t'')$  then there exists  $t < t' < t''$  such that  $P'$  takes a step at time  $t'$ ,  $(P, t) \rightarrow (P', t')$ , and  $(P', t') \rightarrow (P'', t'')$ . Conversely, if  $(P, t) \rightarrow (P', t')$  and  $(P', t') \rightarrow (P'', t'')$ , then  $(P, t) \rightarrow (P'', t'')$ .

**Proposition 2.** Suppose  $P$  takes no steps in the interval  $[t+1, T]$  and  $P'$  takes no steps in the interval  $[t'+1, T']$ , where  $t \leq T$  and  $t' \leq T'$ . Then the following are equivalent:  $(P, t) \rightarrow (P', t')$ ,  $(P, t) \rightarrow (P', T')$ ,  $(P, T) \rightarrow (P', t')$ , and  $(P, T) \rightarrow (P', T')$ .

A subset of the configurations of an algorithm is *closed* if any action performed from a configuration in this set results in a configuration in this set. Let  $H$  be a predicate defined on configurations. An algorithm *stabilizes to  $H$  under a set of schedules  $S$*  if there is a closed set of configurations,  $L$ , all of which satisfy

$H$ , such that every execution whose schedule is in  $S$  contains a configuration that is in  $L$ . The configurations in  $L$  are called *safe*. When an execution reaches a safe configuration, we say that it has stabilized. The *stabilization time* of an algorithm is the maximum, over all executions in  $S$ , of the number of actions performed until the execution stabilizes. A self-stabilizing algorithm is *silent* if no processors are enabled in safe configurations.

Let  $LE$  be the predicate, defined on configurations of an algorithm, that is true when exactly one processor is a leader (i.e. its state is in the specified set). An algorithm that stabilizes to  $LE$  under the set of all fair schedules is called a *self-stabilizing leader election algorithm*. Notice that, once an execution of a leader election algorithm stabilizes, the leader does not change. This is because processors change state one at a time, so between a configuration in which one processor is the only leader and a configuration in which another processor is the only leader, there must be an unsafe configuration.

We present an algorithm in Section 3 that stabilizes to  $LE$  under the set of alternating schedules, defined in Section 2.1, using  $5n$  states per processor. Beauquier, Gradinariu, and Johnen [4] prove the following result about stabilization under the set of alternating schedules.

**Theorem 1.** *Any algorithm on a ring that stabilizes to predicate  $H$  under the set of alternating schedules can be converted into an algorithm that stabilizes to  $H$  under all fair schedules, using only double the number of states (i.e. only one additional bit of storage) at each processor.*

Applying their transformation to our algorithm gives a self-stabilizing leader election algorithm that uses  $10n$  states per processor.

## 2.1 Alternating Schedules

A schedule is *alternating* if, between every two successive steps of each processor, there is exactly one step of its left neighbour and exactly one step of its right neighbour. Any round robin schedule is alternating. For a ring of size 5 with processors  $P_1, P_2, P_3, P_4, P_5$  in order around the ring, the finite schedule  $P_1, P_2, P_5, P_4, P_1, P_5, P_3, P_4, P_2, P_3, P_1, P_2, P_5, P_1, P_4, P_5, P_3, P_2$  is also an alternating schedule. It is equivalent to say that a schedule is alternating if, between every two steps of each processor, there is at least one step of each of its neighbours.

The assumption of an alternating schedule allows us to determine more situations where the state of a processor at one step influences the state of another processor at some later step. The proofs of the following lemmas are by induction on the distance from  $P$  to  $Q$  and can be found in the complete paper. They will be used in the proof that our algorithm stabilizes to  $LE$  under the set of alternating schedules.

The first result says that if a step of  $P$  influences a step of  $Q$ , then earlier or later steps of  $P$  influence correspondingly earlier or later steps of  $Q$ .

**Lemma 1.** *Consider an alternating schedule in which processor  $P$  takes  $k$  steps in the interval  $[t + 1, T]$  and processor  $Q$  takes  $k$  steps in the interval  $[t' + 1, T']$ . Then  $(P, t) \rightarrow (Q, t')$  if and only if  $(P, T) \rightarrow (Q, T')$ .*

Another important property is that each step of each processor will eventually influence some step of every other processor.

**Lemma 2.** *Let  $P$  and  $Q$  be distinct processors. Suppose  $P$  takes a step at time  $t$  of an alternating schedule. Then there exists a step  $t' > t$  of  $Q$  such that  $(P, t) \rightarrow (Q, t')$ .*

The next results bound when a processor will influence another processor, as a function of the distance from one to the other.

**Lemma 3.** *Let  $P$  and  $Q$  be distinct processors, where the distance from  $P$  to  $Q$  is  $k$ . If  $P$  takes at least  $k + 1$  steps by time  $t'$  in an alternating schedule, then there exists a time  $t < t'$  such that  $(P, t) \rightarrow (Q, t')$ ,  $P$  takes at most  $k$  steps in the interval  $[t + 1, t']$ , and  $P$  takes a step at time  $t$ .*

**Lemma 4.** *Let  $P$  and  $Q$  be distinct processors, where the distance from  $P$  to  $Q$  is  $k$ . If  $Q$  takes at least  $k$  steps by time  $t'$  in an alternating schedule, then there exists a time  $t < t'$  such that  $(P, t) \rightarrow (Q, t')$ ,  $Q$  takes at most  $k$  steps in the interval  $[t, t']$ , and either  $t = 0$  or  $P$  takes a step at time  $t$ .*

Finally, the difference between the numbers of steps that have been taken by two processors can be bounded by the distance from one processor to the other.

**Lemma 5.** *Suppose the distance from  $P$  to  $Q$  is  $k$  and  $(P, t) \rightarrow (Q, t')$ . If  $Q$  takes at least  $m$  steps by time  $t'$  in an alternating schedule, then  $P$  takes at least  $m - k$  steps by time  $t$ .*

**Lemma 6.** *Suppose the distance between  $P$  and  $Q$  is  $k$ . If  $P$  takes  $m$  steps by time  $t$ , then  $Q$  takes between  $m - k$  and  $m + k$  steps by time  $t$ .*

### 3 A New Leader Election Algorithm

In this section, we present a deterministic leader election algorithm for a uni-directional ring that uses  $5n$  states per processor and stabilizes under the set of alternating schedules within  $O(n^3)$  time. We begin by describing some of the ideas from Burns and Pachl's token circulation algorithm [7] and how they are used in our leader election algorithm.

In a token circulation algorithm, some, but not all, of the tokens must disappear when more than one token exists. Similarly, in a leader election algorithm, when a ring contains more than one leader, some, but not all, of the leaders must become nonleaders. Because the only direct flow of information is from a processor to the processor on its right, the first token or leader a processor can receive information from is the first one that is encountered travelling left from

the processor. We call this the *preceding* token or leader. The *following* token or leader is the closest one to the processor's right. We define the *strength* of a token or leader to be the distance to it from the preceding token or leader. If there is only one token or leader in a ring of size  $n$ , then its strength is  $n$ .

In Burns and Pachl's basic algorithm, each processor has two variables: one to store its distance from the preceding token (which, in the case of a processor with a token is the strength of that token) and the other to store the strength of the preceding token. The value of each variable is in  $[1, n]$ . Thus, the total number of states per processor is  $O(n^2)$ .

A processor whose left neighbour has a token knows that it is at distance 1 from the preceding token. Any other processor can determine its distance from the preceding token by adding 1 to the corresponding value of its left neighbour. Every processor can obtain the strength of the preceding token directly from its left neighbour: from the first variable, if its left neighbour has a token and from the second variable, if its left neighbour does not have a token.

When a processor with a token learns that the preceding token is stronger, it destroys its own token. On a ring whose size is prime, the distances between successive tokens cannot be all identical. Thus, extra tokens will eventually disappear.

In our algorithm, the state of each processor consists of two components: a *tag*  $X \in \{c, d, B, C, D\}$  and a *value*  $v \in [1, n]$ . We say that a processor is a *leader* if its tag is  $B$ ,  $C$ , or  $D$ ; otherwise it is called a *nonleader*.

The safe configurations of our algorithm each contain exactly one leader, which is in state  $(D, n)$ . In addition, the nonleader at distance  $i$  from the leader is in state  $(d, i)$ , for  $i = 1, \dots, n - 1$ . An example of a safe configuration is illustrated in Figure 1(a). It is easy to verify that no processors are enabled in safe configurations. Hence, our algorithm is silent.

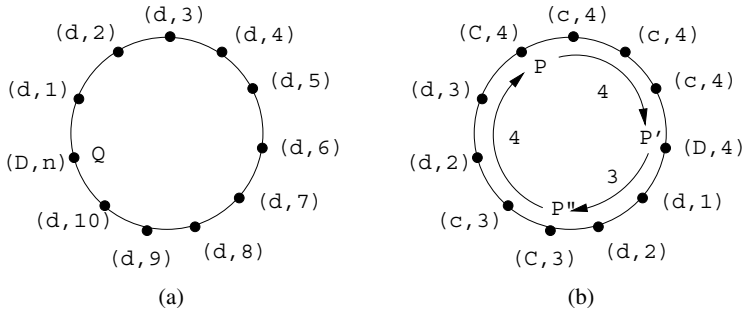
Actions of our algorithm are described by specifying the state  $(X, v)$  of a processor  $P$  and the state  $(X_L, v_L)$  of its left neighbour  $P_L$  and then giving  $P$ 's new state  $(X', v')$ . Such an action will be written  $(X_L, v_L) (X, v) \mapsto (X', v')$ . Instead of presenting all the actions at once, we present them in small groups, together with a brief discussion of their intended effect.

For a leader,  $v$  usually contains its *strength*, i.e. its distance from the preceding leader. Sometimes, nonleaders are used to *determine* the strength of a leader. In this case, the processor has tag  $d$  and  $v$  contains its *distance* from the preceding leader. A nonleader can also have tag  $c$ , indicating that  $v$  is *conveying* strength information from the preceding leader to the following leader.

An example of an unsafe configuration of our algorithm is illustrated in Figure 1(b). Here  $P$  and  $P'$  are leaders with strength 4,  $P''$  is a leader with strength 3, and the other 8 processors are nonleaders.

When the tag of a leader is  $B$  or  $D$ , this signals the sequence of nonleaders to its right to compute their distance from the leader, as follows: The right neighbour of the leader sets its value to 1 and each subsequent nonleader sets its value to one more than the value of the nonleader to its left. They set their





**Fig. 1.** Safe and unsafe configurations of our algorithm

tags to  $d$ , to relay the signal. Because the schedule is alternating, the signal is guaranteed to reach the entire sequence of nonleaders to the right of the leader.

1.  $(X_L, v_L) (X, v) \mapsto (d, 1)$  for  $X_L = B, D$  and  $X = c, d$
2.  $(d, v_L) (X, v) \mapsto (d, 1 + (v_L \bmod n))$  for  $X = c, d$  and  $v_L \neq n - 1$

When the tag of a leader is  $C$ , this signals the nonleaders to its right to convey the strength of this leader, by copying the value from their left neighbour and setting their tag to  $c$ . For example, in Figure 1(b), if the processor at distance 2 from  $P''$  takes the next step, its state will change from  $(d, 2)$  to  $(c, 3)$ .

3.  $(X_L, v_L) (d, v) \mapsto (c, v_L)$  for  $X_L = c, C$

Our algorithm ensures that if a processor has tag  $C$  or  $c$  immediately before it takes a step, then it has neither tag immediately afterwards. This implies that processors with tag  $c$  will not perform either of the following two actions after taking their first step. When the tag of its left neighbour is  $C$ , processor  $P$  with tag  $c$  simply treats its left neighbour's tag as if it were  $D$  and enters state  $(d, 1)$ . However, when its left neighbour's tag is  $c$ , it is possible that all processors have tag  $c$ . To ensure that the ring contains at least one leader,  $P$  becomes a leader.

4.  $(C, v_L) (c, v) \mapsto (d, 1)$
5.  $(c, v_L) (c, v) \mapsto (B, 1)$

A processor in state  $(d, n - 1)$  suggests that its right neighbour is the only leader. If that right neighbour is a nonleader, it can correct the problem by becoming a leader. This avoids another situation where no leader may exist.

6.  $(d, n - 1) (X, v) \mapsto (B, 1)$  for  $X = c, d$

A leader with tag  $B$  is a *beginner*: it has performed  $n$  or fewer steps as a leader. For beginners, the value  $v$  records the number of steps for which the processor has been a leader, up to a maximum of  $n$ . Thus, when a nonleader

becomes a leader, it begins in state  $(B, 1)$ . After a leader has performed  $n$  steps as a beginner, it should get tag  $D$  and record its strength in  $v$ . But when its left neighbour has tag  $c$ , the processor cannot determine its own strength. Therefore it waits in state  $(B, n)$  until its next step. In the meanwhile, its left neighbour will take exactly one step and, hence, will have a different tag. Thus, a processor performs at most  $n + 1$  consecutive steps as a beginner.

7.  $(X_L, v_L) (B, v) \mapsto (B, v + 1)$  for  $X_L = B, c, d$  and  $v \neq n$
8.  $(d, v_L) (B, n) \mapsto (D, 1 + (v_L \bmod n))$
9.  $(B, v_L) (B, n) \mapsto (D, 1)$
10.  $(c, v_L) (B, n) \mapsto (B, n)$

When the system is in a configuration with multiple leaders, some of these leaders have to be destroyed. If  $P$  is a leader whose left neighbour has tag  $C$  or  $D$ , then  $P$  resigns its leadership by setting its state to  $(d, 1)$ . However, if  $P$ 's left neighbour has tag  $B$ , then  $P$  waits in state  $(D, 1)$ . Provided  $P$ 's left neighbour stays a leader long enough,  $P$  will be destroyed, too.

11.  $(X_L, v_L) (X, v) \mapsto (d, 1)$  for  $X_L = C, D$  and  $X = B, C, D$
12.  $(B, v_L) (X, v) \mapsto (D, 1)$  for  $X = C, D$

When  $P$  is a leader whose left neighbour,  $P_L$ , has tag  $c$ , then  $v_L$  contains the strength of the preceding leader. In this case,  $P$  can compare its strength against that of the preceding leader. If  $P$  is at least as strong, it remains a leader. Otherwise,  $P$  resigns its leadership by setting its tag to  $d$ . However, the value  $v_L$  does not provide information from which  $P$  can compute its distance from the preceding leader. Consequently,  $P$  sets its value to  $n$ , to act as a place holder until  $P$ 's next step, when  $P_L$  will have a different tag.

13.  $(c, v_L) (X, v) \mapsto (D, v)$  for  $X = C, D$  and  $v \geq v_L$
14.  $(c, v_L) (X, v) \mapsto (d, n)$  for  $X = C, D$  and  $v < v_L$

A processor can enter state  $(d, n)$  only by resigning its leadership. When the left neighbour of a leader is in state  $(d, n)$ , the leader cannot use  $v_L$  to determine its strength. In this case, the leader leaves its value  $v$  unchanged.

15.  $(d, n) (D, v) \mapsto (D, v)$

When its left neighbour  $P_L$  has tag  $d$ , a leader  $P$  that is not a beginner can update its value with a better estimate of its strength. Such a processor usually alternates its tag between  $C$  and  $D$ . The only exception is when  $P$ 's left neighbour has value  $n - 1$ , which indicates that  $P$  is the only leader. In this case,  $P$  stays in state  $(D, n)$ .

16.  $(d, v_L) (D, v) \mapsto (C, 1 + v_L)$  for  $v_L \neq n - 1, n$
17.  $(d, v_L) (C, v) \mapsto (D, 1 + (v_L \bmod n))$
18.  $(d, n - 1) (D, v) \mapsto (D, n)$

## 4 Properties of the Algorithm

In this section, we present a number of results about the behaviour of our algorithm. They are useful for the proof of correctness presented in Section 5. Throughout this section and Section 5, we assume that all schedules are alternating.

The first result relates the value of a processor with tag  $d$  to its distance from a leader or a processor that has just resigned its leadership. It can be proved by induction.

**Lemma 7.** *Suppose  $(P, t) \rightarrow (Q, t')$ , the distance from  $P$  to  $Q$  is  $k$ , and  $Q$  has tag  $d$  at time  $t'$ . If, at time  $t$ ,  $P$  is a leader or has state  $(d, n)$ , then, at time  $t'$ , either  $Q$  has value  $n$  or value at most  $k$ . Conversely, if  $Q$  has value  $k$  at time  $t'$ , then, at time  $t$ , either  $P$  is a leader or has state  $(d, n)$ .*

A leader that is not a beginner cannot have become a leader recently.

**Lemma 8.** *Let  $[t+1, t']$  be an interval that contains at most  $n$  steps of processor  $P$ . If  $P$  has tag  $C$  or  $D$  at time  $t'$ , then  $P$  is a leader throughout the interval  $[t, t']$ .*

*Proof.* Suppose that  $P$  becomes a leader during the interval  $[t+1, t']$ . At that time,  $P$  has state  $(B, 1)$ . It cannot get tag  $C$  or  $D$  until it has performed at least  $n$  more steps, which occurs after time  $t'$ .

The next result identifies a situation in which a leader cannot be created.

**Lemma 9.** *Suppose  $(P, t) \rightarrow (Q, t')$  and  $Q$  takes at least one step before  $t'$ . If  $P$  is a leader throughout the interval  $[t, t' - 1]$ , then  $Q$  does not become a leader at time  $t'$ .*

*Proof sketch.* Suppose  $Q$  becomes a leader at time  $t'$ . It follows that  $Q_L$  has state  $(d, n-1)$  at time  $t' - 1$ . Then Lemma 7 is applied to obtain a contradiction.

### 4.1 Experienced Leaders

An experienced leader is a processor that has been a leader long enough so that the fact that it is a leader influences and has been influenced by every other processor. Formally, an *experienced leader* is a processor with tag  $C$  or  $D$  that has taken at least  $n$  steps. Then, either an experienced leader has remained a leader since the initial configuration, or it has served its full time as a beginner, since last becoming a leader.

The existence of an experienced leader in a configuration of an execution provides a lot of information about what actions may occur.

**Lemma 10.** *No new leader will be created whenever there is an experienced leader.*

*Proof.* To obtain a contradiction, suppose there is a time  $t''$  at which  $P$  is an experienced leader and  $Q$  becomes a leader. Since  $P$  has taken at least  $n$  steps, it follows by Lemma 6 that  $Q$  has taken at least one step before time  $t''$ . This implies that, at time  $t''$ ,  $Q$  performs action 6,  $Q_L$  has state  $(d, n - 1)$ , and  $P \neq Q, Q_L$ .

From Lemma 3 and Proposition 1, there are times  $t < t' < t''$  such that  $(P, t) \rightarrow (Q_L, t') \rightarrow (Q, t'')$ ,  $Q_L$  takes a step at time  $t'$ , and  $P$  takes at most  $n$  steps in the interval  $[t, t'']$ . Then Lemma 8 implies that  $P$  is a leader throughout this interval. Since the distance from  $P$  to  $Q_L$  is at most  $n - 2$ , it follows from Lemma 7 that  $Q_L$  cannot have value  $n - 1$  at time  $t'$ . However,  $Q_L$  has state  $(d, n - 1)$  at time  $t''$  and takes no steps in the interval  $[t' + 1, t'']$ . Thus  $Q_L$  has state  $(d, n - 1)$  at time  $t'$ . This is a contradiction.

The proofs of the next two results appear in the full paper. They use Proposition 2 and Lemmas 1, 4, 3, 5, 7, 8, and 9.

**Lemma 11.** *If an experienced leader has value  $v > 1$ , then the  $v - 1$  processors to its left are nonleaders.*

This says that the value of an experienced leader is a lower bound on its strength.

**Lemma 12.** *While a processor is an experienced leader, its value never decreases.*

## 5 Proof of Self-Stabilization

Here, we prove that the algorithm presented in Section 3 stabilizes to LE under the set of alternating schedules.

The proof has the following main steps. First, we show that every execution reaches a configuration in which there is a leader. Then, from some point on, all configurations will contain an experienced leader. By Lemma 10, no new leaders will be created, so, eventually, all leaders will be experienced leaders. As in Burns and Pachl's algorithm, if there is more than one leader, they cannot have the same strength, because the ring size is prime. Thus resignations must take place until only one leader remains. Finally, a safe configuration is reached.

**Lemma 13.** *Consider any time interval  $[t, t']$  during which each processor takes at least  $n$  steps. Then there is a time in  $[t - 1, t']$  at which some processor is leader.*

*Proof.* Without loss of generality, we may assume that  $t = 1$ . To obtain a contradiction, suppose that no processor is a leader in  $[0, t']$ . Only actions 2 and 3 are performed during  $[1, t']$ , since all other actions either require or create a leader.

Let  $v$  be the maximum value that any processor has as a result of performing action 2 for its first time. Then  $1 \leq v < n$ . Say processor  $P_0$  has value  $v$  at time  $t_0$  as a result of performing action 2 for its first time. Let  $P_i$  be the processor

at distance  $i$  from  $P_0$ , for  $i = 1, \dots, n - v$ . Then by Lemma 2, there exist times  $t_1 < \dots < t_{n-v}$  such that  $(P_0, t_0) \rightarrow (P_1, t_1) \rightarrow \dots \rightarrow (P_{n-v}, t_{n-v})$  and  $P_i$  takes a step at time  $t_i$  for  $i = 1, \dots, n - v$ . Note that  $P_{n-v}$  takes at most  $n$  steps by time  $t_{n-v}$ ; otherwise, Lemma 5 implies that  $P_0$  takes at least two steps before  $t_0$ . This is impossible, since  $P_0$  cannot perform action 3 twice in a row. Hence,  $t_{n-v} \leq t'$ .

It follows by induction that processor  $P_{n-1-v}$  has state  $(d, n - 1)$  at time  $t_{n-1-v}$ . But then  $P_{n-v}$  performs action 6 at time  $t_{n-v}$  and becomes a leader. This is a contradiction.

**Lemma 14.** *If there is only one experienced leader and it has taken at least  $3n$  steps, then it cannot resign its leadership.*

*Proof sketch.* To obtain a contradiction, suppose that processor  $P_0$  has taken at least  $3n$  steps before time  $t_0$ , it is the only experienced leader at time  $t_0 - 1$ , and it resigns its leadership at time  $t_0$ . Then  $P_0$  performs action 14 at time  $t_0$ . Let  $v_0$  denote the value of  $P_0$  at time  $t_0 - 1$  and let  $k_0 = 0$ .

We prove, by induction, that there exist processors  $P_1, \dots, P_{n-1}$ , values  $v_0 < v_1 < \dots < v_{n-1}$ , distances  $1 < k_1 < \dots < k_{n-1} < n$ , and times  $t_1, t'_1, \dots, t_{n-1}, t'_{n-1}$  such that, for  $i = 1, \dots, n - 1$ ,

- $P_i$  performs action 14 at time  $t_i < t_0$ ,
- $P_i$  takes a step at time  $t'_i < t_i$ ,
- the distance from  $P_i$  to  $P_0$  is  $k_i$ ,
- $P_i$  has value  $v_i$  at time  $t_i - 1$ , and
- $(P_i, t'_i) \rightarrow (P_0, t_0)$ .

But this is impossible.

**Lemma 15.** *After each processor has taken  $6n + 1$  steps, there is always an experienced leader.*

*Proof.* Consider any execution of the algorithm. Let  $t$  be the first time at which every processor has taken at least  $3n$  steps and let  $t'' > t$  be the first time such that all processors have taken at least  $n$  steps in  $[t + 1, t'']$ . By Lemma 13 there is a time  $t' \in [t, t'']$  at which some processor  $P$  is a leader. If  $P$  has tag  $C$  or  $D$  at time  $t'$ , then  $P$  is an experienced leader. Otherwise,  $P$  has state  $(B, v)$  for some value  $v \geq 1$ . Unless an experienced leader is created,  $P$  will perform action 7 at each step until it has state  $(B, n)$ , it will perform action 10 at most once, and then perform action 8 or 9 to become an experienced leader. Note that, at  $t'$ , every processor has taken at least  $3n$  steps, so Lemma 14 implies that there is at least one experienced leader in every subsequent configuration.

By time  $t$ , some processor has taken exactly  $3n$  steps, so Lemma 6 implies that no processor has taken more than  $\lceil 7n/2 \rceil$  steps. Similarly, some processor takes exactly  $n$  steps in the interval  $[t + 1, t'']$ , so no processor takes more than  $\lceil 3n/2 \rceil$  steps in this interval. This implies that  $P$  takes at most  $5n$  steps by time  $t''$ . Therefore  $P$  becomes an experienced leader within  $6n + 1$  steps, since  $P$  takes at most  $n + 1$  steps after  $t'$  until this happens.

**Lemma 16.** *After each processor has taken at least  $\lfloor 15n/2 \rfloor + 2$  steps, all leaders are experienced.*

*Proof.* Consider any execution of the algorithm and let  $t$  be the first time at which every processor has taken at least  $6n+1$  steps. By Lemma 15, the execution contains an experienced leader at all times from  $t$  on. Lemma 10 implies that no new leader will be created after time  $t$ . Any beginner at  $t$  will become an experienced leader or resign its leadership by the time it has taken  $n+1$  more steps. By time  $t$ , some processor has taken exactly  $6n+1$  steps, so Lemma 6 implies that no processor has taken more than  $\lfloor 13n/2 \rfloor$  steps. Thus, by the time each processor has taken  $\lfloor 15n/2 \rfloor$  steps, all leaders are experienced.

**Lemma 17.** *Consider any interval  $[t, t']$  during which the set of leaders does not change, all leaders are experienced, and every processor performs at least  $n+1$  steps. Then, at  $t'$ , the value of every leader is equal to its strength.*

*Proof.* Let  $Q$  be a leader with value  $v$  at  $t'$  and let  $P$  be the processor such that the distance from  $P$  to  $Q$  is  $v$ . Suppose  $T'$  is the last time at or before  $t'$  at which  $Q$  takes a step and  $Q_L$  has tag  $d$ . Processor  $Q_L$  has value  $v-1$  at  $T'$ . By Lemma 3, there is a time  $T < T'$  such that  $(P, T) \rightarrow (Q_L, T')$ ,  $P$  takes at most  $v-1$  steps in  $[T+1, T']$ , and  $P$  takes a step at time  $T$ . Lemma 7 implies that, at time  $T$ , either  $P$  is a leader or has state  $(d, n)$ .

If  $P$  has state  $(d, n)$  at time  $T$ , then  $P$  must have performed action 14 at time  $T$ , resigning its leadership. But  $t < T < T' \leq t'$  and the set of leaders doesn't change throughout the interval  $[t, t']$ . Thus  $P$  is a leader at time  $T$  and, hence, at time  $t'$ .

By Lemma 11, the  $v-1$  processors to the left of  $Q$  are nonleaders at  $t'$ . Hence, at  $t'$ , processor  $P$  is the leader preceding  $Q$  and  $Q$  has strength  $v$ .

**Lemma 18.** *Let  $t$  be any time at which there is more than one leader and all leaders are experienced. If every processor takes at least  $\lfloor 5n/2 \rfloor + 2$  steps in  $[t, t'']$ , then some processor resigns during  $[t+1, t'']$ .*

*Proof.* To obtain a contradiction, suppose that, during  $[t, t'']$ , all processors take at least  $\lfloor 5n/2 \rfloor + 2$  steps and the set of leaders does not change. Let  $t' > t$  be the first time such that every processor performs at least  $n+1$  steps in  $[t, t']$ . Then Lemma 17 implies that, throughout  $[t', t'']$ , the value of every leader is equal to its strength. Let  $P$  be one leader, let  $P'$  be the following leader, and let  $v$  and  $v'$  denote their respective strengths. The processor that takes step  $t'$  takes exactly  $n$  steps during the interval  $[t, t'-1]$ . Then Lemma 6 implies that  $P$  takes at most  $\lfloor 3n/2 \rfloor$  steps during  $[t, t']$ .

Consider the first time  $T \geq t'$  at which  $P$  has tag  $C$ . Then  $P$  performs at most 2 steps in  $[t'+1, T]$ . By Lemma 2, there exist times  $T < T_1 < \dots < T_{v'-1} < T'$  such that  $(P, T) \rightarrow (P_1, T_1) \rightarrow \dots \rightarrow (P_{v'-1}, T_{v'-1}) \rightarrow (P', T')$ , where  $P_i$  is the processor at distance  $i$  from  $P$ . At time  $T_i$ , processor  $P_i$  performs action 3 and gets state  $(c, v)$ , for  $i = 1, \dots, v'-1$ .

From Lemma 3 we know that  $P$  takes at most  $v' \leq n - 1$  steps in  $[T + 1, T']$ . Hence  $P$  takes at most  $\lfloor 5n/2 \rfloor + 1$  steps during  $[t, T']$ . Therefore  $T' < t''$ . Since no processor resigns during  $[t + 1, t'']$ , processor  $P'$  performs action 13 at time  $T'$ . Therefore  $v' \geq v$ .

Since  $P$  is an arbitrary leader, this implies that the strengths of all leaders are the same. Hence, the ring size  $n$  is divisible by the number of leaders. This is impossible, because  $n$  is prime and the number of leaders lies strictly between 1 and  $n$ .

**Lemma 19.** *From any configuration in which there is only one leader and that leader is experienced, the algorithm in Section 3 reaches a safe configuration within  $O(n^2)$  steps.*

*Proof.* Consider a time  $t$  at which there is only one leader  $P$  and suppose  $P$  is experienced at  $t$ . By Lemmas 2 and 3, there exists a step  $t'$  of  $P_L$  such that  $(P, t) \rightarrow (P_L, t')$  and  $P$  takes at most  $n$  steps in  $[t, t']$ . If  $P_L$  has tag  $d$  at time  $t'$ , let  $T' = t'$ ; otherwise, let  $T'$  be the time of  $P_L$ 's next step. Then  $P_L$  has tag  $d$  at time  $T'$ . By Lemma 11, there exists a time  $T$  such that  $(P, T) \rightarrow (P_L, T')$  and, by Lemma 7,  $P_L$  will have value  $n - 1$  at time  $T'$ .

Processor  $P$  gets state  $(D, n)$  at its first step following  $T'$ . From then on,  $P$  remains in state  $(D, n)$ , performing only actions 18 and 13. Lemma 7 and an easy induction on  $k$  show that if the distance from  $P$  to  $Q$  is  $k$ ,  $P$  has state  $(D, n)$  at time  $t''$ , and  $(P, t'') \rightarrow (Q, T'')$ , then  $Q$  has state  $(d, k)$  at time  $T''$ . Thus, within  $O(n^2)$  steps, a safe configuration is reached.

Our main result follows directly from these lemmas.

**Theorem 2.** *The algorithm in Section 3 stabilizes to LE within  $O(n^3)$  steps under any alternating schedule.*

## 6 Conclusion

We have presented a deterministic, self-stabilizing leader election algorithm for unidirectional, prime size rings of identical processors, proved it correct, and analyzed its complexity. The number of states used by this algorithm is  $10n$  per processor, matching the lower bound [4] to within a small constant factor. Combined with Gouda and Haddix's algorithm [13], we get a deterministic, self-stabilizing algorithm for token circulation on unidirectional prime size rings that uses only a linear number of states per processor. This answers the open question in [7] of determining the space complexity of self-stabilizing token circulation on unidirectional rings. We believe our work sheds new insight into the nature of self-stabilization by more precisely delineating the boundary between what is achievable and what is not.

Through our work with alternating schedules, we have improved our understanding of how the state of one processor influences the states of other processors. This enabled us to store two pieces of information alternately in a single

variable, yet have both pieces of information available when needed. This technique may be useful for designing other space efficient self-stabilizing algorithms on the ring and, more generally, on other network topologies.

Our algorithm is silent under an alternating schedule. However, when combined with the deterministic token algorithm, it is not silent: the deterministic tokens circulate forever. One remaining open question is whether there exists a silent, deterministic, self-stabilizing leader election on a unidirectional ring that uses only a linear number of states per processor.

**Acknowledgements.** Faith Fich was supported by grants from the Natural Sciences and Engineering Research Council of Canada and Communications and Information Technology Ontario. Part of this research was done while she was a visitor at Laboratoire de Recherche en Informatique, Université de Paris-Sud.

## References

1. K. Abrahamson, A. Adler, R. Gelbart, L. Higham, and D. Kirkpatrick: The bit complexity of randomized leader election on a ring. *SIAM J. Comput.* **18** (1989) 12–29
2. Dana Angluin: Local and global properties in networks of processors. 12th ACM Symposium on the Theory of Computing. (1980) 82–93
3. Hagit Attiya and Jennifer Welch: *Distributed Computing, Fundamentals, Simulations and Advanced Topics*. McGraw-Hill. (1998)
4. J. Beauquier, M. Gradinariu, and C. Johnen: Memory space requirements for self-stabilizing leader election protocols. *Proceedings of the Eighteenth Annual ACM Symposium on Principles of Distributed Computing*. (1999) 199–208
5. J. Beauquier, M. Gradinariu, and C. Johnen: Self-stabilizing and space optimal leader election under arbitrary scheduler on rings. Internal Report, LRI, Université de Paris-Sud, France. (1999)
6. J. Beauquier, M. Gradinariu, and C. Johnen: Cross-over composition—enforcement of fairness under unfair adversary. *Fifth Workshop on Self-Stabilizing Systems*. (2001)
7. J.E. Burns and J. Pachl: Uniform self-stabilizing rings. *ACM Transactions on Programming Languages and Systems*. **11** (1989) 330–344
8. S. Dolev, M.G. Gouda, and M. Schneider: Memory requirements for silent stabilization. *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*. (1996) 27–34
9. E.W. Dijkstra: Self stabilizing systems in spite of distributed control. *CACM* **17** (1974) 643–644
10. E.W. Dijkstra: Self stabilizing systems in spite of distributed control. *Selected Writing on Computing: A Personal Perspective*. Springer-Verlag. (1982) 41–46
11. E.W. Dijkstra: A belated proof of self-stabilization. *Distributed Computing*. **1** (1986) 5–6
12. Shlomi Dolev: *Self-Stabilization*. MIT Press. (2000)
13. M.G. Gouda and F.F. Haddix: The stabilizing token ring in three bits. *Journal of Parallel and Distributed Computing*. **35** (1996) 43–48
14. Ted Herman: Comprehensive self-stabilization bibliography. <http://www.cs.uiowa.edu/ftp/selfstab/bibliography/> (2001)



15. L. Higham and S. Myers: Self-stabilizing token circulation on anonymous message passing. *Distributed Computing: OPODIS '98*. Hermes. (1998) 115–128
16. S.T. Huang: Leader election in uniform rings. *ACM Transactions on Programming Languages and Systems*. **15** (1993) 563–573
17. G. Itkis, C. Lin, and J. Simon: Deterministic, constant space, self-stabilizing leader election on uniform rings. *Proceedings of the 9th International Workshop on Distributed Algorithms*, Lecture Notes in Computer Science. Springer-Verlag. **972** (1995) 288–302
18. C. Lin and J. Simon: Observing self-stabilization. *Proceedings of the Eleventh Annual ACM Symposium on Principles of Distributed Computing*. (1992) 113–123
19. Nancy Lynch: *Distributed Algorithms*. Morgan Kaufmann. (1996)

# Randomized Finite-State Distributed Algorithms as Markov Chains

Marie Dufлот, Laurent Fribourg, and Claudine Picaronny

LSV, CNRS & ENS de Cachan,  
61 av. du Prés. Wilson,  
94235 Cachan cedex, France  
{duflot,fribourg,picaro}@lsv.ens-cachan.fr

**Abstract.** Distributed randomized algorithms, when they operate under a memoryless scheduler, behave as finite Markov chains: the probability at  $n$ -th step to go from a configuration  $x$  to another one  $y$  is a constant  $p$  that depends on  $x$  and  $y$  only. By Markov theory, we thus know that, no matter where the algorithm starts, the probability for the algorithm to be after  $n$  steps in a “recurrent” configuration tends to 1 as  $n$  tends to infinity. In terms of *self-stabilization* theory, this means that the set  $\mathcal{Rec}$  of recurrent configurations is included into the set  $\mathcal{L}$  of “legitimate” configurations. However in the literature, the convergence of self-stabilizing randomized algorithms is always proved in an elementary way, without explicitly resorting to results of Markov theory. This yields proofs longer and sometimes less formal than they could be. One of our goals in this paper is to explain convergence results of randomized distributed algorithms in terms of Markov chains theory.

Our method relies on the existence of a non-increasing measure  $\varphi$  over the configurations of the distributed system. Classically, this measure counts the number of tokens of configurations. It also exploits a function  $D$  that expresses some distance between tokens, for a fixed number  $k$  of tokens. Our first result is to exhibit a sufficient condition Prop on  $\varphi$  and  $D$  which guarantees that, for memoryless schedulers, every recurrent configuration is legitimate. We extend this property Prop in order to handle arbitrary schedulers although they may induce non Markov chain behaviours. We then explain how Markov’s notion of “lumping” naturally applies to measure  $D$ , and allows us to analyze the expected time of convergence of self-stabilizing algorithms. The method is illustrated on several examples of mutual exclusion algorithms (Herman, Israeli-Jalfon, Kakugawa-Yamashita).

## 1 Introduction

A randomized distributed system is a network of  $N$  finite-state processes whose states are modified via randomized local actions. A process  $P$  is *enabled* when its state and the states of its neighbours allow  $P$  to be the siege of some action. A *scheduler* is a mechanism which, at each step, selects a subset of enabled processes: all the selected processes then execute synchronously an action and

change their state accordingly. A scheduler is *memoryless* when the choice of the enabled processes depends only on the current network configuration  $x$ , i.e. the current value of the  $N$ -tuple of process states. Distributed randomized algorithms, when they operate under a memoryless scheduler, behave as finite Markov chains: the probability at  $n$ -th step to go from a configuration  $x$  to another one  $y$  is a constant  $p$  that depends on  $x$  and  $y$  only. By Markov theory, we thus know that, no matter where the algorithm starts, the probability for the algorithm to be after  $n$  steps in a “recurrent” configuration tends to 1 as  $n$  tends to infinity. In terms of *self-stabilization* theory [56], this means that the set  $\mathcal{R}ec$  of recurrent configurations is included into the set  $\mathcal{L}$  of “legitimate” configurations. However, in the literature (see, e.g., [10,11,9,12,13]) the convergence of self-stabilizing randomized algorithms is often proved in an elementary way, without explicitly resorting to results of Markov theory. This yields proofs longer and sometimes less formal than they could be. One of our goals in this paper is to explain convergence results of randomized distributed algorithms in terms of Markov chains theory.

We focus here on  $\varphi$ -algorithms where  $\varphi$  is a measure over configurations that characterizes the set  $\mathcal{L}$  of legitimate states, and never increases along any computation. For example, for mutual exclusion algorithms, a typical measure  $\varphi$  counts the number of “tokens”, the legitimate configurations being those with only one token. Our first contribution is to exhibit a general property of  $\varphi$ -algorithms, called Prop, which guarantees that, under a memoryless scheduler, every recurrent configuration is legitimate ( $\mathcal{R}ec \subseteq \mathcal{L}$ ). We also show that this property Prop extends naturally to guarantee the convergence of distributed algorithms under arbitrary schedulers, although these algorithms may not behave any longer as Markov chains. Finally we explain how to use the “lumping” method of Markov theory in order to derive a simpler distributed algorithm, which allows us to compute the expected time of convergence of the original algorithm. This gives us a formal justification for the method used, e.g., by Herman [10].

The plan of the paper is as follows. After some preliminaries (Section 2), we define sufficient property Prop that ensures convergence of distributed algorithms under memoryless scheduler (Section 3). Then Prop is extended in order to treat arbitrary schedulers in Section 4. We explain how to use the lumping method in order to analyze the time of convergence in Section 5. We conclude in Section 6.

## 2 Preliminaries

For the sake of simplicity we focus in this paper on the simple topology of linear networks, i.e. rings. We also assume that the communication between processes is done through the reading of neighbours’ state.

### 2.1 Randomized Uniform Ring Systems

The following material is borrowed from [13] (cf. [14]). A *randomized uniform ring system* is a triple  $R = (N, \rightarrow, Q)$  where  $N$  is the number of processes in the

system,  $\rightarrow$  is a state transition algorithm, and  $Q$  is a finite set of process states. The  $N$  processes  $P_0, P_1, \dots, P_{N-1}$  form a ring: the fact that there is an edge from  $P_{i-1}$  to  $P_i$  means that  $P_i$  can observe the state  $q_{i-1}$  of  $P_{i-1}$ <sup>1</sup>. Calculations on indices  $i$  of processes are done modulo  $N$ . Let  $Q$  be the state set of  $P_i$ . The system is *uniform* in the sense that  $\rightarrow$  and  $Q$  are common to every process. A *configuration* of  $R$  is an  $N$ -tuple of process states; if the current state of process  $P_i$  is  $q_i \in Q$ , then the configuration of the system is  $x = (q_0, q_1, \dots, q_{N-1})$ . We denote by  $X$  the set of all configurations, i.e.,  $X = Q^N$ . The state transition algorithm  $\rightarrow$  is given as a set of *guarded commands* of the form:

```

IF <guard1> THEN <command1>
IF <guard2> THEN <command2>
...
IF <guardm> THEN <commandm>

```

Here a guard is a predicate of the form  $g(q_{i-1}, q_i)$ . A command modifies the state  $q_i$  of process  $P_i$ . It is either a deterministic action of the form  $q'_i = h(q_{i-1}, q_i)$ , or a probabilistic action of the form  $q'_i = h^\omega(q_{i-1}, q_i)$  with probability  $p^\omega > 0$  ( $\omega \in \{0, 1\}$  and  $\sum_{\omega=0,1} p^\omega = 1$ ). We say that process  $P_j$  is *enabled* if a guard  $g(q_{j-1}, q_j)$  is true. The set of indices of enabled processes of a configuration  $x$  is denoted by  $\mathcal{E}(x)$ . If no process is enabled at configuration  $x$  ( $\mathcal{E}(x) = \emptyset$ ), we say that there is a *deadlock*. Otherwise, a *scheduler*  $\mathcal{A}$ , is a mechanism which selects a nonempty subset  $S$  of  $\mathcal{E}(x)$ . A transition then leads from  $x$  to  $x' = (q'_0, q'_1, \dots, q'_N)$  with

$$q'_j = \begin{cases} q_j & \text{if } j \notin S, \\ r_j & \text{if } j \in S. \end{cases}$$

Here  $r_j$  is the result of executing **command** <sub>$k(j)$</sub> , where  $k(j)$  is the smallest index  $\ell$  such that **guard** <sub>$\ell$</sub>  holds for  $q_{j-1}$  and  $q_j$ . More precisely:  $r_j = h_{k(j)}^{\omega(j)}(q_{j-1}, q_j)$  with probability  $p_{k(j)}^{\omega(j)}$  if **command** <sub>$k(j)$</sub>  is probabilistic;  $r_j = h_{k(j)}(q_{j-1}, q_j)$  with probability 1 otherwise. Such a transition is written  $x \xrightarrow{S} x'$ , or sometimes more simply  $x \rightarrow x'$ . The probability associated to this transition, written  $p(x \xrightarrow{S} x')$ , is always positive. More precisely,  $p(x \xrightarrow{S} x') = \prod_{j \in S^*} p_{k(j)}^{\omega(j)}$  where  $S^*$  is the subset of indices  $j$  of  $S$  such that **command** <sub>$k(j)$</sub>  is probabilistic. Applying  $\rightarrow$   $\ell$  times is written  $\rightarrow^\ell$ . The reflexive transitive closure of  $\rightarrow$  is denoted  $\rightarrow^*$ . If the scheduler  $\mathcal{A}$  always selects exactly one enabled process (i.e.,  $|S|$  is always equal to 1), then it is a *central scheduler*. Otherwise, it is a *distributed scheduler*.

*Example 1.* In Herman's mutual exclusion algorithm [10], the scheduler  $\mathcal{A}$  is distributed and "maximal": at each step of computation  $x \xrightarrow{S} y$ , the set  $S$  of selected processes is exactly the set  $\mathcal{E}(x)$  of *all* the enabled processes.

The set of states is  $Q = \{0, 1\}$ , and the number of processes is odd. The expression  $\bar{q}$  means  $q + 1$  where  $+$  is addition modulo 2. For each word  $u =$

<sup>1</sup> One can also model similarly systems where  $P_i$  observe not only the state of  $P_{i-1}$  but also that of  $P_{i+1}$ , see Israeli-Jalfon's algorithm (example [11]).

IF  $q_i \neq q_{i-1}$  THEN  $q'_i = \bar{q}_i$   
 IF  $q_i = q_{i-1}$  THEN  $q'_i = \bar{q}_i$  with probability  $1/2$   
   or  $q_i$  with probability  $1/2$ .

Without loss of understanding, we will abbreviate henceforth  $R : (N, \rightarrow, Q)$  as  $\rightarrow$ . We now arbitrarily fix a configuration  $x_0$  as the initial configuration and a scheduler  $\mathcal{A}$ . A computation of  $\rightarrow$  under  $\mathcal{A}$  is a (possibly infinite) sequence of configurations  $x_0, x_1, \dots$ , such that  $x_0$  is the initial configuration and  $x_i \xrightarrow{S_i} x_{i+1}$  for all  $i \geq 0$ , where  $S_i$  is the set of processes selected by  $\mathcal{A}$  at  $i$ -th step. Such a computation of  $\rightarrow$  under  $\mathcal{A}$  is not deterministic because of the existence of probabilistic actions. The *computation tree* associated with  $\rightarrow$  under  $\mathcal{A}$  starting from  $x_0$ , denoted by  $T(\mathcal{A}, x_0)$ , is a rooted tree such that:

1.  $x_0$  is the root,
2. every directed path starting from the root corresponds to a possible computation of  $\rightarrow$  under  $\mathcal{A}$ , and
3. every vertex  $v$  is labeled with the probability  $\pi(v)$  that  $\rightarrow$  under  $\mathcal{A}$  follows the path connecting  $x_0$  to  $v$ . (If the path connecting  $x_0$  to  $v$  is of the form  $x_0 \xrightarrow{S_0} x_1 \xrightarrow{S_1} \dots \xrightarrow{S_{k-1}} x_k$ , then  $\pi(v) = \prod_{j=0}^{k-1} p(x_j \xrightarrow{S_j} x_{j+1})$ .)

We consider now  $\varphi$ -algorithms, i.e., distributed algorithms for which there exists a measure  $\varphi$  over configurations that never increases, whatever the randomized actions do. This situation is typical of mutual exclusion algorithms, where  $\varphi$  counts the number of tokens ( $\varphi$  never increases as tokens are never created during algorithm computations, and decrease when tokens collide.) This is also the case of Israeli-Jalfon *directing* protocol (see [12], Section 4.1) where  $\varphi$  is the number of non-directed edges, or Dolev-Israeli-Moran *leader election* protocol (see [7], Section 5.3) where  $\varphi$  is the number of processes that hold 1 in their leader variables. Formally, a measure  $\varphi$  from  $X$  to  $\mathbb{N}$  is *non-increasing* iff:  $x \xrightarrow{S} y$  implies  $\varphi(y) \leq \varphi(x)$ , for all  $x, y \in X$  and every subset  $S$  of enabled processes.

For a  $\varphi$ -algorithm, we define the set  $\mathcal{L} \subseteq X$  of  $\varphi$ -legitimate configurations as:  $\mathcal{L} = \{x \in X \mid \varphi(x) \leq c\}$  where  $c$  is an integer constant. Since  $\varphi$  is non-increasing, it is easy to show that  $\mathcal{L}$  is *closed*, i.e: For any  $\varphi$ -legitimate configuration  $x \in \mathcal{L}$ , any configuration  $y \in X$  and any set  $S$  of enabled processes,  $x \xrightarrow{S} y$  implies  $y \in \mathcal{L}$ .

We assume furthermore that there is *no deadlock* ( $\forall x \in X \ \mathcal{E}(x) \neq \emptyset$ ). In practice, for mutual exclusion algorithms, the no deadlock property often means that there exists always at least one token ( $\forall x \ \varphi(x) \geq 1$ ). The set  $\mathcal{L}$  of  $\varphi$ -legitimate configurations then actually corresponds to configurations with exactly one token (i.e.:  $x \in \mathcal{L}$  iff  $\varphi(x) = c = 1$ ).

Given a scheduler  $\mathcal{A}$ , we are interested in proving the following *convergence* property (see [13,3]): No matter which initial configuration one starts from, the probability that  $\rightarrow$  under  $\mathcal{A}$  reaches a  $\varphi$ -legitimate configuration in a finite number of transitions is 1. Formally, let  $T^{\mathcal{L}}$  be the tree constructed from  $T(\mathcal{A}, x_0)$  by cutting the edges going out of vertices corresponding to  $\varphi$ -legitimate configurations of  $\mathcal{L}$ ; let  $Leaf(\mathcal{L})$  be the set of leaves of  $T^{\mathcal{L}}$ . The convergence condition claims that for all  $x_0 \in X$ :  $\sum_{v \in Leaf(\mathcal{L})} \pi(v) = 1$ .

Expression  $\sum_{v \in Leaf(\mathcal{L})} \pi(v)$  is the *probability of convergence of  $\rightarrow$  under  $\mathcal{A}$* , i.e., the probability, starting from  $x_0$ , to reach a  $\varphi$ -legitimate configuration in a finite number of transitions. It will be abbreviated as:  $Pr(x_0 \xrightarrow{\mathcal{A}} * \mathcal{L})$ , or sometimes more simply as:  $Pr(x_0 \rightarrow * \mathcal{L})$ .

### 3 Randomized Algorithms as Markov Chains

We assume in this section that scheduler  $\mathcal{A}$  is given and is *memoryless*, i.e: for every sequence of transitions  $x_0 \xrightarrow{S_0} x_1 \xrightarrow{S_1} \dots \xrightarrow{S_{\ell-1}} x_{\ell}$ ,  $\mathcal{A}$  deterministically selects a subset  $S_{\ell}$  of enabled processes of  $x_{\ell}$ , which depends on  $x_{\ell}$  only (not on the previous transitions). Since there is no deadlock,  $S_{\ell}$  is nonempty, and the modification of letters at positions of  $S_{\ell}$  randomly changes  $x_{\ell}$  into a set of possible configurations  $x_{\ell+1}^1, x_{\ell+1}^2, \dots$ , with associated probabilities  $p(x_{\ell} \rightarrow x_{\ell+1}^1), p(x_{\ell} \rightarrow x_{\ell+1}^2), \dots$  of sum 1. More precisely, given  $x \in X$ , consider the set  $I_x$  of all the couples  $(y, p)$  of  $X \times [0, 1]$  such that  $x \rightarrow y$  under  $\mathcal{A}$  with probability  $p$ . Then, for all  $x \in X$ ,  $\sum_{(y,p) \in I_x} p = 1$ . So the computation behaves exactly as a Markov chain [14]. Therefore the classical Markov property holds: whatever the initial configuration we start from, the probability to reach a *recurrent* configuration in a finite number of steps is 1. We exploit this result for proving the convergence of  $\varphi$ -algorithms towards  $\mathcal{L}$ , using the fact that, under a certain condition, each recurrent configuration is  $\varphi$ -legitimate. Let us first recall the notions of “recurrence”, “transience” and Markov basic theorem in our context.

**Definition 2.** A configuration  $x$  is *transient* iff  $\exists y \ x \rightarrow^* y \wedge \neg(y \rightarrow^* x)$ .

**Definition 3.** A configuration  $x$  is *recurrent* iff  $x$  is non transient, i.e:

$$\forall y \ x \rightarrow^* y \Rightarrow y \rightarrow^* x.$$

The set of recurrent configurations is denoted  $Rec$ .

Correspondingly, we define  $T^{Rec}$  as a tree constructed from  $T(\mathcal{A}, x_0)$  by cutting the edges going out from vertices corresponding to recurrent configurations of  $Rec$ . Let  $Leaf(Rec)$  be the set of leaves of  $T^{Rec}$ . The expression

$\sum_{v \in \text{Leaf}(\mathcal{R}ec)} \pi(v)$  is the probability, starting from  $x_0$ , to reach a recurrent configuration in a finite number of transitions. It will be abbreviated as  $Pr(x_0 \xrightarrow{A}^* \mathcal{R}ec)$ .

**Theorem 4. (Markov)** *Given a scheduler  $\mathcal{A}$ , we have: for every configuration  $x$ ,  $Pr(x \xrightarrow{A}^* \mathcal{R}ec) = 1$ .*

We now assume given a non-increasing measure  $\varphi$  on  $X$ , and exploit Markov's theorem.

**Definition 5.** *A configuration  $x$  is  $\varphi$ -transient iff  $\exists y \ x \rightarrow^* y \wedge \varphi(y) < \varphi(x)$ .*

**Lemma 6.** *For every configuration  $x$ , if  $x$  is  $\varphi$ -transient, then  $x$  is transient.*

*Proof.* Suppose that  $x$  is  $\varphi$ -transient. Therefore  $x \rightarrow^* y \wedge \varphi(y) < \varphi(x)$  for some  $y$ . Let us show that  $x$  is transient by proving by reductio ad absurdum that  $\neg(y \rightarrow^* x)$ . If  $y \rightarrow^* x$ , then  $\varphi(x) \leq \varphi(y)$  (since  $\varphi$  is non-increasing), which contradicts  $\varphi(y) < \varphi(x)$ .  $\square$

**Theorem 7.** *Given a scheduler  $\mathcal{A}$  and a non-increasing measure  $\varphi$ , we have: if each configuration  $x$  with  $\varphi(x) > c$  is  $\varphi$ -transient, then:  $\forall x \ Pr(x \xrightarrow{A}^* \mathcal{L}) = 1$ .*

*Proof.* Any non  $\varphi$ -legitimate configuration  $x$  is such that  $\varphi(x) > c$ . So, by assumption, any non  $\varphi$ -legitimate configuration  $x$  is  $\varphi$ -transient. Hence, by Lemma 6,  $x$  is transient, i.e., non recurrent. So:  $\neg \mathcal{L} \subseteq \neg \mathcal{R}ec$ . Hence:  $\mathcal{R}ec \subseteq \mathcal{L}$ . Now, by Markov theorem,  $Pr(x \xrightarrow{A}^* \mathcal{R}ec) = 1$ . It follows:  $Pr(x \xrightarrow{A}^* \mathcal{L}) = 1$ .  $\square$

Let us now give a local condition, called “Prop”, ensuring that every non-legitimate configuration is  $\varphi$ -transient. This condition is local in the sense that it involves only one-step reduction  $\rightarrow$  (instead of  $\rightarrow^*$ ).

We assume given a scheduler  $\mathcal{A}$  and a non-increasing measure  $\varphi$ . For every value of ring length  $N$ , we assume given a measure  $D$  from  $X = Q^N$  to a finite set  $\Delta$  and an ordering  $\ll$  over  $\Delta$ . We then define a binary relation  $\triangleleft$  over  $X$  as follows:

$$\forall x, y \in X \quad x \triangleleft y \iff \varphi(x) < \varphi(y) \vee D(x) \ll D(y).$$

Since  $<$  and  $\ll$  are orderings,  $\triangleleft$  is itself an ordering over the finite set  $X = Q^N$ . Local condition Prop is defined as:

$$\forall x \quad (\varphi(x) > c \Rightarrow \exists y \ (x \rightarrow y \wedge x \triangleleft y)).$$

It says that, for each configuration  $x$  (of  $\varphi$ -measure  $> c$ ), there exists a transition going from  $x$  to a configuration  $y$  smaller w.r.t.  $\triangleleft$ . As there is no infinite decreasing sequence for  $\triangleleft$  (since  $\Delta$  is finite), Prop ensures the  $\varphi$ -transience of each non-legitimate configuration. Therefore by Theorem 7 it follows:

**Theorem 8.** *Given a scheduler  $\mathcal{A}$  and a non-increasing measure  $\varphi$ , we have: if, for all  $N$ , there exist a measure  $D$  and an ordering  $\ll$  such that Prop:*

$$\forall x \quad (\varphi(x) > c \Rightarrow \exists y \ (x \rightarrow y \wedge (\varphi(x) < \varphi(y) \vee D(x) \ll D(y))))$$

*holds, then:  $\forall x \ Pr(x \xrightarrow{A}^* \mathcal{L}) = 1$ .*

Given  $\mathcal{A}$  and  $\varphi$ , the crucial part of the convergence proof consists now in finding appropriate  $D$  and  $\ll$  that satisfy Prop. In practice,  $D$  and  $\ll$  will not be defined specifically for each ring length  $N$ , but in a generic manner with  $N$  as a parameter.

*Example 9.* We apply here Theorem 8 for showing that Herman's algorithm is convergent. It is easy to notice (see [10]) that, starting from a configuration with an odd number of processes,  $\rightarrow$  has no deadlock. Given a configuration  $x = q_0 q_1 \cdots q_{N-1}$  we say there is a token at process  $i$  if  $q_i = q_{i-1}$ . The measure  $\varphi$  counts the number of tokens of a configuration  $x$ , i.e:  $\varphi(x) = \text{card}(\{i \in \{0, \dots, N-1\} \mid q_i = q_{i-1}\})$ . It is also proved in [10] that  $\varphi$  is non-increasing. The set  $\mathcal{L}$  of  $\varphi$ -legitimate configurations is defined as the set of configurations  $x$  with at most one token ( $\varphi(x) \leq c = 1$ ).

For configurations  $x$  with at least two tokens, we consider the same measure  $D$  as the one introduced by Herman [10], i.e. the minimal distance between two consecutive tokens of  $x$ . The corresponding ordering  $\ll$  then coincides with  $<$ . For example, for  $x = \mathbf{010100101101010}$ , there are three tokens (represented in bold font), and the minimal distance between them is  $D(x) = 4$ . In the following, we assume that  $u$  and  $v$  are strings of  $Q^*$ ,  $a$  and  $b$  are distinct elements of  $Q$ . If  $x$  has a pair of adjacent tokens (i.e.,  $x$  is of the form  $uaaav$ ), then  $D(x) = 1$ . Otherwise, for some  $n \geq 0$ ,  $x$  may be of the form  $uaa(ba)^{n-1}baav$  with  $D(x) = 2n + 1$  (odd case) or of the form  $uaa(ba)^nbbv$ , thus  $D(x) = 2n + 2$  (even case).

Let us show that, for all  $x$  with at least two tokens ( $\varphi(x) > 1$ ), there exists  $y$  such that  $x \rightarrow y$  with  $\varphi(y) < \varphi(x)$  or  $D(y) < D(x)$ . There are three cases:

- If  $D(x) = 1$ ,  $x$  is of the form  $uaaav$  and  $x \rightarrow y = \bar{u}bab\bar{v}$  with  $\varphi(y) = \varphi(x) - 2$ .
- If  $D(x) = 2n + 1$  with  $n \geq 0$ ,  $x$  is of the form  $uaa(ba)^{n-1}baav$  and  $x \rightarrow y = \bar{u}ba(ab)^{n-1}abb\bar{v}$  with  $D(y) = 2n < D(x)$ .
- If  $D(x) = 2n + 2$  with  $n \geq 0$ ,  $x$  is of the form  $uaa(ba)^nbbv$  and  $x \rightarrow y = \bar{u}ba(ab)^n aa\bar{v}$  with  $D(y) = 2n + 1 < D(x)$ .

By Theorem 8 it follows that:  $\forall x \Pr(x \rightarrow^* \mathcal{L}) = 1$ . This proof is much simpler than the original one of [10]. Many other examples can be treated exactly along these lines (e.g., Beauquier-Delaët [2], Flatebo-Datta [9]).

## 4 Randomized Algorithms under Arbitrary Scheduler

Let us consider now the case where the scheduler  $\mathcal{A}$  is *not* memoryless, but has unlimited resources, and chooses the next enabled processes using the full information of the execution so far. In such a case, for the same configuration  $x$ , the selection of enabled processes can vary with the different sequences of transitions that led to  $x$ . The computation under such a scheduler  $\mathcal{A}$  does not behave any longer as a Markov chain. Property Prop of Theorem 8 (dependent on a specific scheduler) must be strengthened in order to take into account all possible schedulers: see Prop' below. For the sake of simplicity, we focus on the case where the scheduler is central (the selected subset of enabled processes is a singleton). The counterpart of Theorem 8 is the following:



**Theorem 10.** *Given a non-increasing measure  $\varphi$ , suppose that there exist  $D$  and  $\ll$  such that:*

*Prop':  $\forall x \forall i \in \mathcal{E}(x) \quad (\varphi(x) > c \Rightarrow \exists y (x \xrightarrow{i} y \wedge (\varphi(x) < \varphi(y) \vee D(x) \ll D(y))))$*

*Then, for any central scheduler  $\mathcal{A}$ :  $\forall x \quad \Pr(x \xrightarrow{\mathcal{A}}^* \mathcal{L}) = 1$ .*

The proof is given in appendix A. The result extends to the distributed case in the natural way (by replacing  $\mathcal{E}(x)$  with  $2^{\mathcal{E}(x)}$ ). Theorem 10 can be seen as a restricted version of Theorem 1 of [3] (see also theorem 5 of [7]). In [8], we prove the convergence of Kakugawa-Yamashita's algorithm [13] using Theorem 10. For lack of space, we present below an application to a simpler algorithm.

*Example 11.* Let us consider Israeli-Jalfon's algorithm [11]. The scheduler is central and arbitrary. A minor difference with the framework presented in Section 2.1 is that each guard takes into account not only the state  $q_{i-1}$  of the left neighbour of  $P_i$ , but also the state  $q_{i+1}$  of the right neighbour. Also each command modifies not only the state  $q_i$  of  $P_i$ , but also the states of the left and right neighbours. (There is no conflict between transitions because the scheduler is central, and only one enabled process  $P_i$  is selected at each step.) The set of states is  $Q = \{0, 1\}$ . The transition algorithm  $\rightarrow$  is:

IF  $q_{i-1}q_iq_{i+1} = 111$  THEN  $q'_{i-1}q'_iq'_{i+1} = 101$

IF  $q_{i-1}q_iq_{i+1} = 011$  THEN  $q'_{i-1}q'_iq'_{i+1} = 101$  with probability  $1/2$   
or  $001$  with probability  $1/2$ .

IF  $q_{i-1}q_iq_{i+1} = 110$  THEN  $q'_{i-1}q'_iq'_{i+1} = 101$  with probability  $1/2$   
or  $100$  with probability  $1/2$ .

IF  $q_{i-1}q_iq_{i+1} = 010$  THEN  $q'_{i-1}q'_iq'_{i+1} = 100$  with probability  $1/2$   
or  $001$  with probability  $1/2$ .

Given a configuration  $x$  we say there is a token at process  $i$  if  $q_i = 1$ . The measure  $\varphi$  counts the number of tokens of a configuration  $x$ . We focus on initial configurations  $x$  with at least one token ( $\varphi(x) > 0$ ). Then all the subsequent configurations keep always at least one token and  $\rightarrow$  has no deadlock. It is also obvious that  $\varphi$  is non-increasing (no '1' is created). The set  $\mathcal{L}$  of  $\varphi$ -legitimate configurations is defined as the set of configurations  $x$  with at most one token ( $\varphi(x) \leq c = 1$ ).

Given a configuration with a fixed number of tokens, say  $k > 1$ , we consider the measure  $D$  that maps any configuration  $x$  with the  $k$ -tuple of distances between two tokens ordered by increasing order. For example, for  $x = 000110001010$ ,  $k = 4$  and  $D(x) = (1, 2, 4, 5)$ .

Let us show that, for all  $x$  with at least two tokens ( $\varphi(x) > 1$ ), and every position  $i$  of token (or '1') in  $x$ , there exists  $y$  such that  $x \xrightarrow{i} y$  with  $\varphi(y) < \varphi(x)$  or  $D(y) \ll D(x)$ , where  $\ll$  is the lexicographic order. There are four cases:

- If  $q_{i-1}q_iq_{i+1} = 111$ , then  $x$  is of the form  $u111v$  and  $x \xrightarrow{i} y = u101v$  with  $\varphi(y) = \varphi(x) - 1$ .
- If  $q_{i-1}q_iq_{i+1} = 011$ , then  $x$  is of the form  $u011v$  and  $x \xrightarrow{i} y = u001v$  (with probability  $1/2$ ) with  $\varphi(y) = \varphi(x) - 1$ .

- If  $q_{i-1}q_iq_{i+1} = 110$ , then  $x$  is of the form  $u110v$  and  $x \xrightarrow{i} y = u100v$  (with probability  $1/2$ ) with  $\varphi(y) = \varphi(x) - 1$ .
- If  $q_{i-1}q_iq_{i+1} = 010$ , then  $x$  is of the form  $u010v$ . Let  $\ell$  (resp.  $r$ ) be the distance between the token at position  $i$  and the closest token at his left (resp. right). Such a left (resp. right) closest token exists because  $\varphi(x) > 1$ . (Note that the left and right closest tokens coincide if  $\varphi(x) = 2$ .) If  $\ell \leq r$  then  $x \xrightarrow{i} y = u100v$  (with probability  $1/2$ ) with  $D(y) \ll D(x)$ . Otherwise,  $x \xrightarrow{i} y = u001v$  (with probability  $1/2$ ) with  $D(y) \ll D(x)$ .

By Theorem 10, it follows that, for any scheduler  $\mathcal{A}$ :  $\forall x \Pr(x \xrightarrow{\mathcal{A}}^* \mathcal{L}) = 1$ .

## 5 Expected Time of Convergence by $D$ -Lumping

We assume again in this section that scheduler  $\mathcal{A}$  is given and is *memoryless*. So the computation via  $\rightarrow$  under  $\mathcal{A}$  behaves exactly as a Markov chain [14]. Let  $k$  be an integer greater than  $c$ ,  $X_k$  the set of configurations of  $\varphi$ -measure  $k$ , and  $X_{<k}$  the set of configurations of  $\varphi$ -measure less than  $k$ . Let  $X_k^d$  be the subset of configurations of  $X_k$  of  $D$ -measure  $d$ , and  $\Delta_k$  the image of  $X_k$  via  $D$ . We have:

$$\begin{aligned} X_k &= \{x \in X \mid \varphi(x) = k\}. \\ X_{<k} &= \{x \in X \mid \varphi(x) < k\}. \\ X_k^d &= \{x \in X \mid \varphi(x) = k \wedge D(x) = d\}. \\ \Delta_k &= \{d \in \Delta \mid \exists x \in X_k \ D(x) = d\}. \end{aligned}$$

Note that  $X_{<k}$  is closed under  $\rightarrow$  (since  $\varphi$  is non-increasing). We assume:

$$\forall x \in X_k \ \exists y \in X_{<k} \quad x \rightarrow^* y.$$

This means that for every configuration  $x$  with  $k$  tokens ( $k > c$ ), there exists a computation that goes from  $x$  to a configuration with less than  $k$  tokens. Condition Prop of Theorem 8 is a sufficient condition that guarantees the existence of such a computation. We want now to get quantitative information about the expected time of convergence of  $\rightarrow$  under  $\mathcal{A}$  from  $x_0 \in X_k$  to  $X_{<k}$ . Formally, given  $x_0 \in X_k$ , let  $T^{X_{<k}}$  be the tree obtained from  $T(\mathcal{A}, x_0)$  by cutting the edges going out from vertices corresponding to configurations of  $X_{<k}$ . Let  $Leaf(X_{<k})$  be the set of leaves of  $T^{X_{<k}}$ . We are interested in computing (an upper bound for) the expected time of  $x_0$  to reach  $X_{<k}$ , that is  $\sum_{v \in Leaf(X_{<k})} \delta(v) \pi(v)$

where  $\delta(v)$  and  $\pi(v)$  are the depth and probability of  $v$  in  $T^{X_{<k}}$  respectively.

This will be abbreviated as  $E(x_0 \xrightarrow{\mathcal{A}}^* X_{<k})$ , or more simply as:  $E(x_0 \rightarrow^* X_{<k})$ . We now explain how to compute this quantity, under certain conditions, by “lumping” using measure  $D$ . In the Markov theory, it is indeed common to lump together configurations, in order to get a smaller Markov chain which gives information about the original chain. Given  $x \in X_k$  and  $e \in \Delta_k$ , consider the expression:

$$\xi(x, e) = \sum_{y \in X_k^e} p(x \rightarrow y).$$

This is the probability of moving via  $\rightarrow$  under  $\mathcal{A}$  from an element  $x$  of  $X_k$  into the set  $X_k^e$  of configurations of  $D$ -measure  $e$ . Likewise, consider the expression:

$$\xi(x, \perp) = \sum_{y \in X_{<k}} p(x \rightarrow y),$$

where  $\perp$  is a new symbol. This is the probability of moving via  $\rightarrow$  under  $\mathcal{A}$  from

an element  $x$  of  $X_k$  into set  $X_{<k}$  of configurations of  $\varphi$ -measure less than  $k$ . Finally, let:

$$\xi(\perp, \perp) = 1.$$

This expresses the fact that  $X_{<k}$  is a set which once entered is never left. We say that Markov chain  $\rightarrow$  is *D-lumpable* if, for all  $x \in X_k$  and all  $e \in \Delta_k \cup \{\perp\}$ , probability  $\xi(x, e)$  depends only on the  $D$ -measure  $d$  of  $x$ , i.e:

$$\forall d \in \Delta_k \forall e \in \Delta_k \cup \{\perp\} \forall x, x' \in X_k^d \quad \xi(x, e) = \xi(x', e).$$

We then write such a probability  $\xi(d, e)$ . Given a  $D$ -lumpable transition system  $\rightarrow$ , the associated *D-transition system*, written  $\rightsquigarrow$ , is the binary relation over  $\Delta_k \cup \{\perp\}$  defined as follows:

- for all  $d, e \in \Delta_k$ ,  $d \rightsquigarrow e$  with probability  $\xi(d, e)$ .
- for all  $d \in \Delta_k$ ,  $d \rightsquigarrow \perp$  with probability  $\xi(d, \perp)$ ,
- $\perp \rightsquigarrow \perp$  with probability  $\xi(\perp, \perp) = 1$ .

By Markov theory, if  $\rightarrow$  is a lumpable Markov chain, then the lumped transition system  $\rightsquigarrow$  is also a Markov chain. The *D-computation tree* associated with  $\rightsquigarrow$  starting from  $d_0 \in \Delta_k \cup \{\perp\}$ , denoted by  $U(\mathcal{A}, d_0)$ , is a rooted tree such that:

1.  $d_0$  is the root,
2. every directed path starting from the root corresponds to a possible sequence of transitions via  $\rightsquigarrow$ , and
3. every vertex  $w$  is labeled with the probability  $\psi(w)$  corresponding to the path connecting  $d_0$  to  $w$ . (If the path connecting  $d_0$  to  $w$  is of the form  $d_0 \rightsquigarrow d_1 \rightsquigarrow \dots \rightsquigarrow d_\ell$ , then  $\psi(w) = \prod_{j=0}^{\ell-1} \xi(d_j, d_{j+1})$ .)

Let  $U^\perp$  be the tree constructed from  $U(\mathcal{A}, d_0)$  by cutting the edges going out from vertices corresponding to  $\perp$ . Let  $Leaf(\perp)$  be the set of leaves of  $U^\perp$ . Let  $\psi(w)$  and  $\varepsilon(w)$  be the probability and the depth of  $w$  respectively, for every vertex  $w$  in  $U^\perp$ . The expression  $\sum_{w \in Leaf(\perp)} \varepsilon(w) \psi(w)$  is the expected number of  $D$ -transitions, starting from  $d_0$ , to reach  $\perp$ . It will be abbreviated as:  $E_k(d_0 \rightsquigarrow^* \perp)$ . We now explain, by Markov theory, how to compute this quantity and how it relates to  $E(x_0 \rightarrow^* X_{<k})$ .

The *D-transition matrix* is the square matrix of size  $(|\Delta_k| + 1)$  having  $\xi(d, e)$  on the row and column corresponding to  $d$  and  $e$  respectively, for all  $d, e \in \Delta_k \cup \{\perp\}$ .

**Definition 12.** An absorbing element  $\alpha$  is an element such that, for all  $d \in \Delta_k \cup \{\perp\}$ ,  $\alpha \rightsquigarrow^* d \Rightarrow d = \alpha$ . The set of absorbing elements is denoted by  $Abs$ . A Markov chain is absorbing iff all recurrent elements are absorbing and conversely ( $Rec = Abs$ ).

Note that, by definition, all absorbing elements are recurrent ( $Abs \subseteq Rec$ ). Furthermore, in our case, it is clear that  $\perp \in Abs$  (since  $\perp \rightsquigarrow \perp$  with probability 1). Besides, from the assumption that:  $\forall x \in X_k \exists y \in X_{<k} \quad x \rightarrow^* y$ , it follows by lumpability that:  $\forall d \in \Delta_k \quad d \rightsquigarrow^* \perp$  (convergence towards  $\perp$ ). Markov theory states the following (see [14], p.59) :

**Theorem 13. (Markov2)** *For every absorbing Markov chain, we have:  $E_k(d \rightsquigarrow^* \text{Abs}) < \infty$ . Furthermore,  $(E_k(d \rightsquigarrow^* \text{Abs}))_{d \in \Delta_k} = (\mathcal{I} - \mathcal{Q}_k)^{-1} \mathbf{1}$ , where  $\mathcal{Q}_k$  is the matrix obtained by truncating the  $D$ -transition matrix onto the set of non-absorbing elements,  $\mathcal{I}$  the identity matrix of size  $|\Delta_k|$ , and  $\mathbf{1}$  is the column matrix made of  $|\Delta_k|$  elements 1.*

Besides, we have:

**Lemma 14.** *For the  $D$ -transition system,  $\text{Abs} = \text{Rec} = \{\perp\}$ . So the  $D$ -transition system is an absorbing Markov chain.*

*Proof.* We have always  $\text{Abs} \subseteq \text{Rec}$ . Let us prove  $\text{Rec} \subseteq \text{Abs}$  by showing that if  $d \notin \text{Abs}$ , then  $d \notin \text{Rec}$ . Suppose  $d \notin \text{Abs}$ . Then  $d \neq \perp$ . Hence  $\neg(\perp \rightsquigarrow^* d)$ . On the other hand, by convergence towards  $\perp$ , we have:  $d \rightsquigarrow^* \perp$ . Since  $d \rightsquigarrow^* \perp$  and  $\neg(\perp \rightsquigarrow^* d)$ ,  $d \notin \text{Rec}$ .  $\square$

From Theorem 13 and Lemma 14, it follows:

**Corollary 15.** *For all  $d \in \Delta_k$ ,  $E_k(d \rightsquigarrow^* \perp) < \infty$ . Furthermore,  $(E_k(d \rightsquigarrow^* \perp))_{d \in \Delta_k} = (\mathcal{I} - \mathcal{Q}_k)^{-1} \mathbf{1}$ , where  $\mathcal{Q}_k$  is the matrix obtained by truncating the  $D$ -transition matrix onto  $\Delta_k$  (i.e, removing the  $\perp$ -row and  $\perp$ -column).*

In addition, since the original Markov chain  $\rightarrow$  is  $D$ -lumpable, Markov theory says that the lumped  $D$ -transition system  $\rightsquigarrow$  is such that (see 14):

$$\forall d \in \Delta_k \ \forall x \in X_k^d \quad E(x \rightarrow^* X_{<k}) = E_k(d \rightsquigarrow^* \perp).$$

It is interesting to compute  $E_k(d \rightsquigarrow^* \perp)$  rather than  $E(x \rightarrow^* X_{<k})$  because the transition matrix of the lumped Markov chain is much smaller than the matrix of the original chain. For example, in Herman's example in case  $k = 2$ , the matrix of the lumped chain is of size  $\lfloor N/2 \rfloor$  while the original one is of size  $2^N$ . The computation of  $E_k(d \rightsquigarrow^* \perp)$  in Herman's example is explained below.

*Example 16.* Let us first show that the Markov chain corresponding to Herman's algorithm is  $D$ -lumpable for  $k = 2$ . This is due to the invariance by rotation of the minimal distance of configurations. Formally let  $x$  be a given configuration with 2 tokens of minimal distance  $D(x) = d$ . Let  $i$  be the position of the first token, and  $j$  the position of the second one. We write  $x = (i, j)$ . The distance  $d$  is  $\min(i - j, j - i)$  where  $-$  is subtraction modulo  $N$ , and ranges over  $\Delta_2 = \{1, 2, \dots, \lfloor N/2 \rfloor\}$  (as  $N$  is odd,  $\lfloor N/2 \rfloor = (N - 1)/2$ ). For  $1 < d < \lfloor N/2 \rfloor$ ,  $x$  moves either to  $x_0 = (i, j)$ ,  $x_1 = (i + 1, j + 1)$ ,  $x_2 = (i + 1, j)$  or  $x_3 = (i, j + 1)$  with equal probability  $1/4$ . Therefore,  $\xi(x, d) = 1/2 (= p(x \rightarrow x_0) + p(x \rightarrow x_1))$ , and  $\xi(x, d + 1) = \xi(x, d - 1) = 1/4$ . If  $d = 1$ , then  $\xi(x, 1) = 1/2$ ,  $\xi(x, 2) = 1/4$  and  $\xi(x, \perp) = 1/4$  (case where the 2 tokens collide). If  $d = (N - 1)/2$ ,  $\xi(x, d) = 3/4$  and  $\xi(x, d - 1) = 1/4$ . Given  $d$  and  $e$ , the value of each probability  $\xi(x, e)$  is constant, whatever the choice of  $x \in X_2^d$ . Hence the lumpability property.

Let us now explain the computation of the  $D$ -transition matrix  $\mathcal{Q}_k$  for  $k = 2$  in Herman's example 10.  $\Delta_2 = \{1, 2, \dots, m\}$  with  $m = N/2$ .  $\mathcal{Q}_2$  is the  $m \times m$  matrix of components  $\xi(d, e)$  (with  $d, e \in \Delta_2$ ), of the form:

$$\begin{pmatrix} 1/2 & 1/4 & & & & \\ 1/4 & 1/2 & 1/4 & & & \\ & & \cdot & \cdot & \cdot & \\ & & & \cdot & \cdot & \cdot \\ & & & & 1/4 & 1/2 & 1/4 \\ & & & & & 1/4 & 3/4 \end{pmatrix}$$

Note that, component  $\xi(1, \perp) = 1/4$  is excluded from the matrix  $Q_2$ , since the  $\perp$ -column has been truncated. We then compute  $B_2 = (\mathcal{I} - Q_2)^{-1}$ , which gives:

$$\begin{pmatrix} 4 & 4 & \cdot & \cdot & \cdot & 4 \\ 4 & 8 & 8 & \cdot & \cdot & 8 \\ \cdot & 8 & 12 & \cdot & \cdot & 12 \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 4 & 8 & 12 & \cdot & \cdot & 4m \end{pmatrix}$$

By Corollary [15](#), we know that the result of applying  $B_2$  to  $\mathbf{1}$  gives a column vector of  $d$ -component  $2d(N - d)$ , for  $d \in \{1, \dots, \lfloor N/2 \rfloor\}$ . Therefore  $E_2(d \rightsquigarrow^* \perp) = 2d(N - d)$ . The maximal expected time corresponds to  $d = \lfloor N/2 \rfloor = m$ , and is  $2m(m + 1) \simeq N^2/2$ . This corresponds to  $E(x \rightarrow^* \mathcal{L})$  for  $x \in X_2$ . We thus retrieves directly what Herman obtained in [\[10\]](#) in a more complicated way. Using this result, Herman explains how to infer an expected time  $N^2 \lceil \log N \rceil / 2$  for  $E(x \rightarrow^* \mathcal{L})$  in the general case where  $x \in X_k$  with  $k > 2$  (see [\[10\]](#)). Israeli-Jalfon's algorithm is analyzed similarly in appendix B.

## 6 Final Remarks and Further Work

We exploited Markov chains theory in order to simplify the proofs of convergence of randomized self-stabilizing algorithms and justify more formally their performance analysis. Our method relies on the existence of a non-increasing measure  $\varphi$  over the configurations of the distributed system. Classically, this measure counts the number of tokens of configurations. It also exploits a function  $D$  that expresses some distance between tokens, for a fixed number  $k$  of tokens. Our first result was to exhibit a sufficient condition Prop that exploits  $\varphi$  and  $D$  in order to guarantee that, under a memoryless scheduler, every non-legitimate configuration is “transient” in the sense of Markov theory. Roughly speaking, Prop says that, under a given scheduler  $\mathcal{A}$ , for every configuration  $x$ , there exists a transition of non-null probability that applies to  $x$  and decreases  $\varphi$  or  $D$ . We extended this property Prop in order to handle arbitrary schedulers although they may induce non Markov chain behaviours. We thus retrieve (a particular case of) a result due to [\[3\]](#). We then explain how Markov's notion of lumping naturally applies to measure  $D$ , and allows to analyze the expected time of

convergence of self-stabilizing algorithms. Let us point out that the crucial step for proving convergence of such distributed algorithms is still the discovery of an appropriate function  $D$  satisfying Prop or Prop' (which may be intricate, see e.g., example of Kakugawa-Yamashita in [8]), but our work identifies some weak properties of  $D$  which suffice to entail the algorithm convergence and justify the performance analysis as done, e.g., by Herman. We only treated the case of finite-state reading model of communication as well as ring topologies. The finite-state assumption is a basic requirement of our work, which seems difficult to be relaxed. On the other hand, we believe that our results apply to other network topologies than rings. In the future, we plan to compare our lumping-based analysis method with the scheduler-luck game technique of [7].

**Acknowledgements.** We are most grateful to Joffroy Beauquier and anonymous referees for helpful comments on the earlier version of this paper.

## References

1. J. Beauquier, S. Cordier, and S. Delaët. Optimum probabilistic self-stabilization on uniform ring. In *Proc. of the 2nd Workshop on Self-Stabilizing Systems*, 1995.
2. J. Beauquier and S. Delaët. Probabilistic self-stabilizing mutual exclusion in uniform ring. In *Proc. of the 13th Annual ACM Symposium on Principles of Distributed Computing (PODC'94)*, page 378, 1994.
3. J. Beauquier, J. Durand-Lose, M. Gradinariu, and C. Johnen. Token based self-stabilizing uniform algorithms. *J. of Parallel and Distributed Systems*, To appear.
4. J. Beauquier, M. Gradinariu, and C. Johnen. Memory space requirements for self-stabilizing leader election protocols. In *Proc. of the 18th Annual ACM Symposium on Principles of Distributed Computing (PODC'99)*, pages 199–208, 1999.
5. E.W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, Nov. 1974.
6. S. Dolev. *Self-Stabilization*. MIT Press, 2000.
7. S. Dolev, A. Israeli, and S. Moran. Analyzing expected time by scheduler-luck games. *IEEE Transactions on Software Engineering*, 21(5):429–439, May 1995.
8. M. DufLOT, L. Fribourg, and C. Picaronny. Randomized distributed algorithms as markov chains. Technical report, Lab. Specification and Verification, ENS de Cachan, Cachan, France, May 2001. Available on <http://www.lsv.ens-cachan.fr/Publis/RAPPORTS.LSV/>.
9. M. Flatebo and A.K. Datta. Two-state self-stabilizing algorithms for token rings. *IEEE Transactions on Software Engineering*, 20(6):500–504, June 1994.
10. T. Herman. Probabilistic self-stabilization. *IPL*, 35(2):63–67, June 1990.
11. A. Israeli and M. Jalfon. Token management schemes and random walks yield self-stabilizing mutual exclusion. In *Proc. of the 9th Annual ACM Symposium on Principles of Distributed Computing (PODC'90)*, pages 119–131, 1990.
12. A. Israeli and M. Jalfon. Uniform self-stabilizing ring orientation. *Information and Computation*, 104(2):175–196, 1993.
13. H. Kakugawa and M. Yamashita. Uniform and self-stabilizing token rings allowing unfair daemon. *IEEE Trans. Parallel and Distributed Systems*, 8(2), 1997.
14. J.G. Kemeny and J.L. Snell. *Finite Markov Chains*. D. van Nostrand Co., 1969.

## Appendix A: Proof of Theorem 10

Before proving this theorem, it is convenient to introduce the notion of “ $\triangleleft$ -decreasing sequence” and some related properties.

**Definition 17.** Given  $x \in X$ , a sequence  $\sigma = (i_1, \dots, i_\ell)$  of process indices is said to be  $\triangleleft$ -decreasing for  $x$  iff:

$$x = x_0 \xrightarrow{i_1} x_1 \xrightarrow{i_2} \dots \xrightarrow{i_\ell} x_\ell \quad \text{for some } x_1, \dots, x_\ell \in X \text{ such that}$$

- $x_\ell \triangleleft x_{\ell-1} \triangleleft \dots \triangleleft x_1 \triangleleft x_0$ , or
- $x_i \triangleleft x_{i-1} \triangleleft \dots \triangleleft x_1 \triangleleft x_0$  and  $\varphi(x_i) \leq c$ , for some  $i$  ( $0 \leq i < \ell$ ).

For any integer  $\ell$ , the finite set of  $\triangleleft$ -decreasing sequences for  $x$  of length  $\ell$  is written  $\text{Dec}_\ell(x)$ .

We will write henceforth  $x \xrightarrow{\sigma} \mathcal{L}$  with  $\sigma = (i_1, \dots, i_\ell)$  as an abbreviation of:  $\exists x_1, \dots, x_\ell \quad x \xrightarrow{i_1} x_1 \wedge x_1 \xrightarrow{i_2} x_2 \wedge \dots \wedge x_{\ell-1} \xrightarrow{i_\ell} x_\ell \in \mathcal{L}$ .

**Lemma 18.** Relation  $\triangleleft$  is an ordering over  $X = Q^N$ . Furthermore, there exists  $M > 0$  such that every sequence of configurations decreasing for  $\triangleleft$  is of length  $\leq M$ .

*Proof.* First, it is easy to see that  $\triangleleft$  is an ordering because  $<$  and  $\ll$  are orderings. Let us now show that, for  $m \geq M = |X|$ , any sequence  $x_0, \dots, x_m$  of elements in  $X$  cannot be ordered by  $\triangleleft$ . By the pigeonhole principle, there exists  $i, j$  with  $0 \leq i < j \leq m$  such that  $x_i = x_j$ . Hence, we cannot have  $x_m \triangleleft \dots \triangleleft x_j \triangleleft \dots \triangleleft x_i \triangleleft \dots \triangleleft x_0$ .  $\square$

**Lemma 19.** For  $M$  defined as above, we have:  $\forall x \in X \quad \forall \sigma \in \text{Dec}_M(x) \quad x \xrightarrow{\sigma} \mathcal{L}$ .

*Proof.* By definition, given  $x \in X$ , for all  $\sigma = (i_1, \dots, i_M) \in \text{Dec}_M(x)$ , there exist  $x_1, \dots, x_M$  such that  $x = x_0 \xrightarrow{i_1} x_1 \xrightarrow{i_2} \dots \xrightarrow{i_M} x_M$  with

$$x_M \triangleleft x_{M-1} \triangleleft \dots \triangleleft x_1 \triangleleft x_0, \quad \text{or}$$

$$x_i \triangleleft x_{i-1} \triangleleft \dots \triangleleft x_1 \triangleleft x_0 \quad \text{and} \quad \varphi(x_i) \leq c, \quad \text{for some } i \quad (0 \leq i < M).$$

But the first case  $x_M \triangleleft x_{M-1} \triangleleft \dots \triangleleft x_1 \triangleleft x_0$  is impossible according to Lemma 18. So:  $\varphi(x_i) \leq c$ , for some  $i$  ( $0 \leq i < M$ ). Hence  $x \xrightarrow{\sigma} \mathcal{L}$ .  $\square$

### Proof of Theorem 10:

Given  $x \in X$  and  $\sigma$  such that  $x \xrightarrow{\sigma} \mathcal{L}$ , let  $\text{Pr}(x \xrightarrow{\sigma} \mathcal{L})$  be the probability associated with  $x \xrightarrow{\sigma} \mathcal{L}$ . Let us now show

$$P_1 : \quad \exists p \in ]0, 1] \quad \forall x \in X \quad \forall \sigma \in \text{Dec}_M(x) \quad \text{Pr}(x \xrightarrow{\sigma} \mathcal{L}) \geq p,$$

By iteratively applying Prop', it is easy to show that, for all  $x \in X$  and all  $\ell \in \mathbb{N}^*$ , there exists a  $\triangleleft$ -decreasing sequence  $\sigma \in \text{Dec}_\ell(x)$ . For all  $x \in X$ , we know by applying Lemma 19 to  $x$  that, for all  $\sigma \in \text{Dec}_M(x)$ ,  $\text{Pr}(x \xrightarrow{\sigma} \mathcal{L}) > 0$ . Since  $\text{Dec}_M(x)$  is nonempty and finite, we can define  $p_x$  as the minimum of  $\text{Pr}(x \xrightarrow{\sigma} \mathcal{L})$  when  $\sigma$  ranges over  $\text{Dec}_M(x)$ . We have:  $\forall \sigma \in \text{Dec}_M(x) \quad \text{Pr}(x \xrightarrow{\sigma} \mathcal{L}) \geq p_x > 0$ .

Now by taking the minimum  $p$  of all the  $p_x$  when  $x$  ranges over  $X$ , we get:  
 $\forall x \in X, \forall \sigma \in Dec_M(x) \ Pr(x \xrightarrow{\sigma} \mathcal{L}) \geq p > 0$ . Hence  $P_1$ .

Observe that, by Prop', for every configuration  $x$  and every scheduler  $\mathcal{A}$ , there is a path in the computation tree  $T(\mathcal{A}, x)$  along which the probabilistic actions make the successive configurations decrease for  $\triangleleft$ . So from  $P_1$  it follows that the probability, starting from  $x$ , to reach  $\mathcal{L}$  under  $\mathcal{A}$  in  $M$  transitions is  $\geq p$ . This writes:  $Pr(x \xrightarrow{\mathcal{A}}^M \mathcal{L}) \geq p$ . Alternatively, given a scheduler  $\mathcal{A}$  and a starting configuration  $x$ , the probability of not being in  $\mathcal{L}$  is less than  $1 - p$  after the  $M$  first transitions. It is less than  $(1 - p)^2$  after the  $2M$  first transitions, and so on. The probability of not reaching  $\mathcal{L}$  under  $\mathcal{A}$  after  $\ell$  transitions tends to 0 as  $\ell$  tends to  $\infty$ . In other words:  $\forall x \in X, \lim_{\ell \rightarrow \infty} Pr(x \xrightarrow{\mathcal{A}}^\ell \mathcal{L}) = 1$ . So, for any central scheduler  $\mathcal{A}$ :  $\forall x \ Pr(x \xrightarrow{\mathcal{A}}^* \mathcal{L}) = 1$ .  $\square$

## Appendix B: Expected Time of Israeli-Jalfon's Algorithm

As mentioned earlier, the scheduler in Israeli-Jalfon's example is arbitrary and the algorithm does not behave *a priori* as a Markov chain. However in the case where there are only two tokens ( $k = 2$ ), one can suppose that the scheduler is memoryless, and always selects the same token, say  $A$ , since any move of the other one can be simulated by a symmetrical move on  $A$ . It is also easy to show that Markov chain corresponding to Israeli-Jalfon's algorithm is  $D$ -lumpable for  $k = 2$ . The  $D$ -chain corresponds to a random walk, and the expected time  $E_2(d \rightsquigarrow^* \perp)$  corresponds to the expected time that one token at a distance  $d$  from the origin, animated by a random walk, meets that origin. For  $N = 2m + 1$ , we have:

$$\mathcal{Q}_2 = \begin{pmatrix} 0 & 1/2 & & & & \\ 1/2 & 0 & 1/2 & & & \\ & \cdot & \cdot & \cdot & & \\ & & \cdot & \cdot & \cdot & \\ & & & \cdot & \cdot & \\ & & & 1/2 & 0 & 1/2 \\ & & & & 1/2 & 1/2 \end{pmatrix}$$

$$(\mathcal{I} - \mathcal{Q}_2)^{-1} = \begin{pmatrix} 2 & 2 & \cdot & \cdot & \cdot & \cdot & 2 \\ 2 & 4 & \cdot & \cdot & \cdot & \cdot & 4 \\ \cdot & \cdot & 6 & \cdot & \cdot & \cdot & 6 \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ 2 & 4 & 6 & \cdot & \cdot & \cdot & 2m \end{pmatrix}$$

Then  $E_2(d \rightsquigarrow^* \perp) = d(N - d)$  with a maximum of  $m(m + 1) \simeq (N/2)^2$  when  $d = m$ . Using our general matrix method, we thus retrieve the quadratic complexity result found by Israeli and Jalfon [11].



# The Average Hop Count Measure for Virtual Path Layouts

## (Extended Abstract)

David Peleg<sup>1\*</sup> and Uri Pincas<sup>2\*\*</sup>

<sup>1</sup> The Weizmann Institute of Science

<sup>2</sup> Bar Ilan University

**Abstract.** This paper studies the average hop count measure for virtual path layouts of ATM and optical networks. Routing in the ATM and optical network models is based on covering the network with simple virtual paths, under some constraints on the allowed *load* (i.e., the number of paths that can share an edge). The *hop count* is the number of edges along the virtual path.

Two basic results are established concerning the average hop count parameter. The first concerns comparing the maximum and average hop count measures assuming uniform all-to-all communication requirements. We develop a rather general connection between the two measures for virtual path layouts with bounded maximum load. This connection allows us to extend known lower bounds on the maximum hop count into ones on the average hop count for network families satisfying certain conditions, termed *non-condensingly contractable (NCC)* graph families. Using this characterization, we establish tight lower bounds on the average hop count of virtual path layouts with bounded maximum load for paths, cycles, and trees.

Our second result is an algorithm for designing a virtual path layout of minimum average hop count for a given tree network with general (weighted) one-to-all requirements.

## 1 Introduction

This paper concerns the problem of designing efficient virtual path layouts on optical or ATM networks (see, e.g., [16,17,20,23]). In the ATM model, the routing and message forwarding tasks on a given network are simplified by predefining a collection of “expressways,” or *virtual paths (VP’s)*, which are simple paths in the network, and performing end-to-end communication over routes composed of a sequence of such VP’s (i.e., using the VP’s as basic segments within complete

---

\* Department of Applied Mathematics and Computer Science, The Weizmann Institute, Rehovot, 76100 Israel. [peleg@wisdom.weizmann.ac.il](mailto:peleg@wisdom.weizmann.ac.il). Supported in part by a grant from the Israel Ministry of Science and Art.

\*\* Department of Mathematics and Computer Science, Bar Ilan University, Ramat Gan 52900, Israel. [uripinch@macs.biu.ac.il](mailto:uripinch@macs.biu.ac.il)

routes). An elegant formulation of the problem can be obtained by representing the VP's formed on the given communication network  $G$  as a virtual graph  $H$  over the same set of vertices. Specifically, a VP connecting  $v$  and  $u$  in  $G$  is represented by an edge connecting  $v$  and  $u$  in the virtual graph  $H$ . The pair  $(H, P)$ , where  $P$  is the collection of VP's corresponding to the edges of  $H$ , is referred to as the *virtual path layout (VPL)* for the physical graph  $G$ . Each route in  $G$  can be viewed as a simple path in the virtual graph  $H$ .

Various formulations of the VPL problem attempt to design a system of virtual paths which optimizes some parameters of the system while meeting some given communication demands between pairs of nodes and satisfying certain prespecified constraints. Usually, the problem is studied against one canonical pattern of communication demands, known as *all-to-all* communication, which requires communication between every pair of vertices in the network.

Research on virtual path layouts has concentrated on optimizing two central parameters of conflicting nature. The first is the *load* of a physical edge, defined as the number of VP's that share it. The upper bound on the load of an edge is termed the *capacity* of the edge. The *maximum (respectively, average) load* of a VPL  $(H, P)$ , denoted  $\mathcal{L}_{max}(H, P)$  (resp.,  $\mathcal{L}_{avg}(H, P)$ ), is defined as the maximum (resp., average) load of the edges in the network. These parameters determine the size of the VP routing tables, and reflect the traffic load on the links.

The second parameter of interest concerns the *hop count*, namely, the number of VP's occurring on the routes of the VPL, or equivalently, the number of edges in the corresponding path on the virtual graph  $H$ . Expressed in terms of  $H$ , the *maximum* hop count, denoted  $\mathcal{H}_{max}(H)$ , can be viewed as the *diameter* of  $H$ , and the *average* hop count, denoted  $\mathcal{H}_{avg}(H)$ , can be defined as the average distance over all vertex pairs in  $H$ . Equivalently, one may consider the *total* number of hop counts over all vertex pairs in  $H$ , termed the *total hop count* of  $H$  and denoted  $\mathcal{H}_{tot}(H)$ . (Clearly,  $\mathcal{H}_{tot}(H) = \mathcal{H}_{avg}(H) \cdot n(n-1)/2$  for every graph  $H$ .) These parameters measure the (worst-case or average) efficiency of setting up the route, and also the overall (worst-case or average) delay incurred by the route in a model where the processing along VP's is negligible compared to the processing at the VP endpoints. See [23] for a discussion of these parameters and their significance.

A number of studies have tackled the VPL problem (cf. [3,19,8,15]). The problem of minimizing  $\mathcal{H}_{max}(H)$ , the diameter of a virtual graph  $H$ , subject to a specified upper bound on the maximum load of the VPL,  $\mathcal{L}_{max}(H, P) \leq c$ , has been considered in the undirected case in [15,14,23,13,18,10]. Conversely, the problem of minimizing the maximum load  $\mathcal{L}_{max}(H, P)$  over all VPL's  $(H, P)$  with bounded maximum hop count,  $\mathcal{H}_{max}(H) \leq h$ , is studied in [11,4]. Minimizing also the average load  $\mathcal{L}_{avg}(H, P)$  is considered in [14].

As links based on optical fibers are directed, and may have a different load in the two directions, it may be useful to consider a directed model as in [7,6], rather than an undirected one. The problem of minimizing the diameter  $\mathcal{H}_{max}(H)$  of a virtual directed graph (digraph)  $H$  over bounded maximum load

VPL's is considered in [6], giving lower and upper bounds on the virtual diameter  $\mathcal{H}_{max}(H)$  of a directed VPL (henceforth, DVPL) with a prespecified capacity  $c$  bounding the maximum load (considered as constant).

This paper focuses on the average hop count measure  $\mathcal{H}_{avg}$  of a digraph, and explores the variant of the VPL problem which seeks to optimize this measure. (Actually, for convenience, we formulate our results in terms of the equivalent *total hop count* measure,  $\mathcal{H}_{tot}(H)$ .) Clearly, the *upper* bounds established in [6] can be converted into ones for the *average* hop count  $\mathcal{H}_{avg}(H)$  of DVPL's as well. However, the average distance may admit better bounds, and hence it is not a-priori clear how to obtain tight or near-tight *lower* bounds for  $\mathcal{H}_{avg}(H)$  over DVPL's with bounded maximum load. In fact, to the best of our knowledge, no such nontrivial lower bounds were known so far.

Our first contribution concerns establishing such lower bounds. In Section 3 we develop a rather general and fundamental connection between the average hop count measure  $\mathcal{H}_{avg}(H)$  and the maximum hop count measure  $\mathcal{H}_{max}(H)$  for DVPL's  $(H, P)$  with bounded maximum load. This connection allows us to extend known lower bounds on  $\mathcal{H}_{max}(H)$  into ones on  $\mathcal{H}_{avg}(H)$  for network families satisfying certain conditions, termed *non-condensingly contractable (NCC)* graph families. Using this characterization, we establish tight lower bounds on the average hop count (or equivalently on the total hop count) of DVPL's with bounded maximum load for a number of network families, including paths, cycles, and trees.

So far, we defined the VPL problem considering an all-to-all pattern of communication demands. Another pattern of communication demands for which the VPL problem was studied is the *one-to-all* pattern, in which a single vertex must communicate with all other vertices [11, 14]. For the chain network, a duality between the problem of minimizing the hop count knowing the maximum load and the one of minimizing the load knowing the maximum hop count, is established in [11]. A number of VPL optimization problems with one-to-all communication demands are studied in [14] for the chain network, including the problem of designing a VPL with optimal average hop count under the one-to-all communication pattern, and a dynamic programming algorithm is presented for this problem. The "Open problems" section of [14] states the following:

"The most immediate open problem is to generalize these results for arbitrary trees, a task which seems non-trivial, as far as the dynamic programming algorithms are concerned, due to the additional structural information that is attached to each subtree (which does not exist in chains)."

Our second contribution, presented in Section 4 involves solving this open problem. As in [14], our solution handles the more general *weighted* version of the problem, in which we are given a *requirements vector*  $\omega$  such that for  $1 < i \leq n$ ,  $\omega_i$  specifies the expected amount of traffic between  $v_1$  and  $v_i$ . Now the weighted average hop count of  $G$  is defined by weighing the distances according to  $\omega$ . We also extend the solution in another way: instead of assuming a uniform constant capacity  $c$  on all links, we allow a somewhat more general capacity function

$c : E \mapsto \mathbb{N}^+$ , under the restriction that  $c(e) \leq c_0$  for every link  $e$ , for some constant  $c_0$ .

## 2 Preliminaries

A physical communication network is represented by an  $n$ -vertex strongly connected directed graph  $G = (V, E)$ . The vertex set  $V$  represents the network switches, and the arc set  $E$  represents the set of physical directed links.

For two vertices  $v, u \in V$  in  $G = (V, E)$ , the *distance* of  $u$  from  $v$  in  $G$ , denoted by  $d_G(v, u)$ , is the length of the shortest path from  $v$  to  $u$  in  $G$ . The *diameter* (or *maximum hop count*) of a graph  $G$  is the largest distance achieved by a pair of its nodes, i.e.,

$$\mathcal{H}_{max}(G) = \max_{(v,u) \in V \times V} \{d_G(v, u)\}.$$

The *total hop count* of  $G$  is defined as

$$\mathcal{H}_{tot}(G) = \sum_{u,v \in V} d_G(u, v).$$

Our main interest is in strongly-connected graphs, where the total hop count is always finite.

For a family of graphs  $\mathcal{M}$ , define  $\mathcal{H}_{max}(\mathcal{M}) = \max_{G \in \mathcal{M}} \{\mathcal{H}_{max}(G)\}$  and  $\mathcal{H}_{tot}(\mathcal{M}) = \max_{G \in \mathcal{M}} \{\mathcal{H}_{tot}(G)\}$ . Let us now give two simple and general upper and lower bounds on  $\mathcal{H}_{tot}(G)$  for an arbitrary digraph  $G$ . (Throughout, most proofs are omitted from this extended abstract.)

**Lemma 1.** *For every  $n$ -vertex digraph  $G$ ,  $n^2 - n \leq \mathcal{H}_{tot}(G) \leq (n^3 - n^2) / 2$ .*

We note that the upper bound is attained by the undirected cycle graph, where the diameter is  $n - 1$ , and each node is an edge node of a diameter path.

One question to reckon with, is the connection between the total hop count of a graph and its diameter. An easy but useful result is the following.

**Lemma 2.** *For every  $n$ -vertex graph  $G$ ,  $n \cdot \mathcal{H}_{max}^2(G)/2 \leq \mathcal{H}_{tot}(G) \leq n^2 \cdot \mathcal{H}_{max}(G)$ .*

Let us now turn to defining the VPL problem and its relevant parameters. Given a network  $G = (V, E)$ , we can assign to certain pairs of distinct vertices  $x, y \in V$  a simple directed path (dipath)  $P(x, y)$ , connecting  $x$  to  $y$ . Consider a set  $E' \subseteq V \times V$  containing every vertex pair  $\langle x, y \rangle$  for which such a dipath  $P(x, y)$  is defined. We consider a new digraph  $H = (V, E')$ , called a *virtualization* of  $G$ . The path  $P(e) = P(x, y)$  in the original graph  $G$  associated with the arc  $e = \langle x, y \rangle$  in  $H$  is called a *virtual path* (VP). Note that  $H$  is not necessarily strongly connected, but we limit our discussion to strongly connected virtualizations, unless stated otherwise. In our terminology, the pair  $(H, P)$  is a *directed virtual path layout* (DVPL) on  $G$ . With each dipath  $Q = (e'_1, \dots, e'_l)$  in  $H$  we associate a *route* in  $G$  consisting of the concatenation of  $P(e'_1), \dots, P(e'_l)$ .

Given a collection  $P$  of virtual paths and an arc  $e$ , let  $T_P[e]$  denote the collection of all virtual dipaths  $P(e')$ , for  $e' \in H$ , that contain the arc  $e$ , that is,  $T_P[e] = \{e' \in E' \mid e \in P(e')\}$ . The *load* of an arc  $e$  of  $G$  is the number of virtual dipaths that contain it, i.e.,  $l(e) = |T_P[e]|$ . The *maximum load* of a DVPL  $(H, P)$  is denoted by  $\mathcal{L}_{\max}(H, P) = \max_{e \in E} \{l(e)\}$ . A DVPL  $(H, P)$  satisfying  $\mathcal{L}_{\max}(H, P) \leq c$  is referred to as a *c-admissible directed virtual path layout* (or a *c-DVPL*) of  $G$ .

Given a graph  $G$  and a positive integer  $c$ , we denote by  $\text{Virt}(G, c)$  the set of all  $c$ -admissible virtualizations of  $G$ . We now define the *minimal realizable diameter* of  $(G, c)$  as the minimal diameter that can be achieved by a virtualization from  $\text{Virt}(G, c)$ , i.e.,

$$\tilde{\mathcal{H}}_{\max}(G, c) = \min\{\mathcal{H}_{\max}(H) \mid H \in \text{Virt}(G, c)\}.$$

Similarly, define the *minimal realizable total hop count* of  $(G, c)$  as the minimal total hop count that can be achieved by such a virtualization, i.e.,

$$\tilde{\mathcal{H}}_{\text{tot}}(G, c) = \min\{\mathcal{H}_{\text{tot}}(H) \mid H \in \text{Virt}(G, c)\}.$$

### 3 The Total Hop Count of Virtual Graphs

In this section we establish upper and lower bounds on  $\tilde{\mathcal{H}}_{\text{tot}}(G_n, c)$  in certain graph families, including paths, cycles, and trees.

One direction is easy; by a direct application of Lemma 2, we can upper bound the total hop count of any graph family in terms of its diameter, as follows.

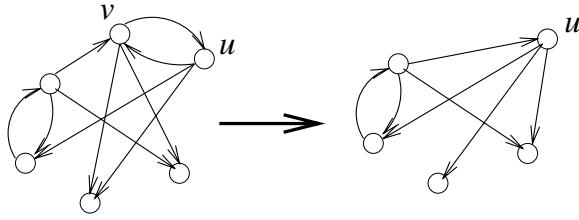
**Lemma 3.** *For every  $n \in \mathbb{N}$  and  $n$ -vertex graph  $G_n$ ,*

$$\tilde{\mathcal{H}}_{\text{tot}}(G_n, c) \leq n^2 \cdot \tilde{\mathcal{H}}_{\max}(G_n, c).$$

Our main goal in this section is to derive *lower bounds* matching the upper bound of the last lemma for various graph classes. This is achieved by developing a general connection between the average hop count measure  $\mathcal{H}_{\text{avg}}(H)$  and the maximum hop count measure  $\mathcal{H}_{\max}(H)$  for DVPL's  $(H, P)$  with bounded maximum load, for network families satisfying certain conditions, termed *non-condensingly contractable (NCC)* graph families.

Let us first introduce the following definitions. Let  $G = (V, E)$  be a (directed or undirected) graph, and let  $(v, u) \in E$  be an edge of  $G$ . The *contraction* of  $v$  to  $u$  is the operation of deleting  $v$  from  $G$ , and reattaching all its edges to  $u$ , namely, replacing every edge  $(v, w)$  with the edge  $(u, w)$ . (In the case of a directed graph, replace each arc  $\langle v, w \rangle$  (respectively  $\langle w, v \rangle$ ) with the arc  $\langle u, w \rangle$  (respectively  $\langle w, u \rangle$ ).) Resulting multiple edges and self loops are removed from the graph. (See Figure 1.)

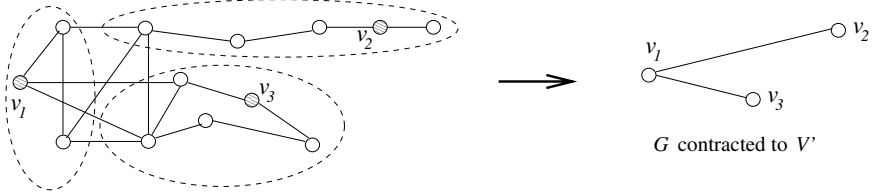
Let  $G = (V, E)$  be a (directed or undirected) connected graph, and let  $V'$  be a subset of  $V$ . A *contraction of  $G$  to  $V'$*  is a function  $\varphi : V \rightarrow V'$  having the following properties:



**Fig. 1.** The contraction of  $v$  to  $u$ .

(C1)  $\varphi(v') = v'$ , for every  $v' \in V'$ .

(C2) For every  $v' \in V'$ , the subgraph  $G(\varphi^{-1}(v'))$  induced by  $\varphi^{-1}(v')$  is connected.



**Fig. 2.** Contracting  $G$  to  $V' = \{v_1, v_2, v_3\}$  according to  $\varphi$ , where  $\varphi^{-1}(v_i)$  is the set enclosed by the dashed ellipse around  $v_i$ , for  $i = 1, 2, 3$ . Vertices of  $V'$  are darkened.

For a contraction  $\varphi$  of  $G$  to  $V'$ , define the graph  $\varphi(G) = (V', E')$ , where

$$E' = \{(v', u') \mid v', u' \in V', \exists v \in \varphi^{-1}(v'), \exists u \in \varphi^{-1}(u') \text{ s.t. } (v, u) \in E\}.$$

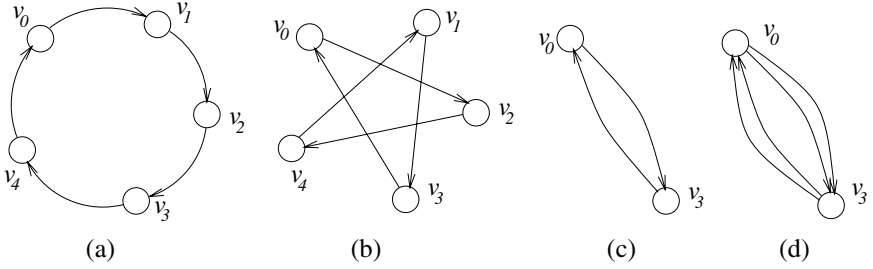
A contraction can be implemented by the following iterative process. Let  $k = 0$  and  $V_0 = V'$ . While  $V \setminus V_k$  is not empty, select (by some fixed rule) a node  $u \in V \setminus V_k$  adjacent to some node  $v \in V_k$ , contract  $v$  to  $u$  (or  $u$  to  $v$ , according to the direction of the arc, if  $G$  is directed), and let  $V_{k+1} = V_k \cup \{u\}$ , and  $k = k + 1$ . Note that the contraction process (and hence its outcome) is not unique, and it is determined by the specific selection rule used. The resulting contraction  $\varphi$  is defined for this process as  $\varphi(v) = v$  for every  $v \in V'$ , and  $\varphi(u) = v$  for every  $u \notin V'$  such that  $v$  is contracted to  $u$  (or  $u$  is contracted to  $v$ ) during the process, as, for some  $k$ ,  $v \in V_k$  and  $u \in V \setminus V_k$ .

Now we consider the induced operation of contraction on virtual graphs. Let  $G = (V, E)$  be a connected graph, and let  $H$  be a virtualization of  $G$ . Let  $V'$  be a subset of  $V$ , and let  $\varphi$  be a contraction of  $G$  to  $V'$ . The *virtualization of  $H$  induced by  $\varphi$* , denoted as  $\mathcal{V}(\varphi, H) = H_\varphi$ , is the virtualization of  $\varphi(G)$  where there is a virtual edge from  $v'$  to  $u'$ , for  $v', u' \in V'$ , iff there is a virtual edge in  $H$  from some node in  $\varphi^{-1}(v')$  to some node in  $\varphi^{-1}(u')$ .

The following example shows that  $H_\varphi$  is not necessarily a legal contraction of  $H$  (even if  $H$  is strongly-connected).

*Example 1.* Let  $G$  be the five clockwise-unidirectional cycle graph, with  $V = \{v_0, v_1, v_2, v_3, v_4\}$  ordered clockwise (see Figure 3(a)). Let  $H$  be a virtualization of  $G$ , with virtual arcs  $(v_i, v_{i+2 \pmod{5}})$ , for  $0 \leq i \leq 4$ . (See Figure 3(b).)

Taking  $V' = \{v_0, v_3\}$ , and  $\varphi(v_0) = \varphi(v_1) = v_0$ ,  $\varphi(v_2) = \varphi(v_3) = \varphi(v_4) = v_3$ , we get a legal contraction of  $G$  to  $V'$  (see Figure 3(c)), but the induced virtualization is *not* a legal contraction of  $H$  to  $V'$  (the subgraph  $H(\varphi^{-1}(v_3))$  induced by  $\varphi^{-1}(v_3)$  is not connected (see Figure 3(d)). ■

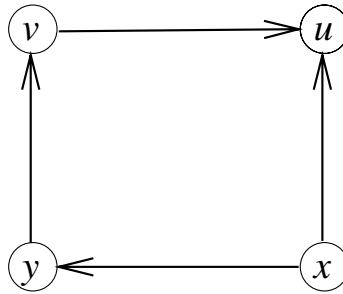


**Fig. 3.** (a) The graph  $G$ . (b) The virtual graph  $H$ . (c)  $\varphi(G)$ , the contraction of  $G$  to  $V' = \{v_0, v_3\}$ . (d) The induced virtual graph  $\varphi(H)$ .

We are interested in a specific type of contractions, which preserve the legality of a  $c$ -DVPL on a given graph. For this we need to introduce some definitions. First, we generalize the definition of an induced virtualization by a contraction to an induced DVPL. Let  $G = (V, E)$  be a graph, and let  $(H, P)$  be a DVPL of  $G$ . Let  $V'$  be a subset of  $V$ , and let  $\varphi$  be a contraction of  $G$  to  $V'$ . The DVPL of  $\varphi(G)$ , defined as follows:  $H_\varphi$  is the virtualization of  $H$  induced by  $\varphi$  (as defined above); each arc of  $H_\varphi$ ,  $e' = (v', u')$ , is defined by a virtual arc of  $H$ ,  $e = (v, u)$ , with  $v \in \varphi^{-1}(v')$  and  $u \in \varphi^{-1}(u')$ . If the dipath  $P(e)$  in  $G$  associated with  $e \in H$  is  $v_1, v_2, \dots, v_k$ , then the dipath  $P_\varphi(e')$  in  $\varphi(G)$  associated with  $e' \in H_\varphi$  is defined to be  $\varphi(v_1), \varphi(v_2), \dots, \varphi(v_k)$ , omitting all self loops.  $P_\varphi$  is taken to be the set of all such dipaths,  $P_\varphi = \{P_\varphi(e') \mid e' \in H_\varphi\}$ . (E.g., in the situation described in Figure 3(b), there is a virtual arc  $(v_4, v_1)$ , whose associated dipath in  $G$  is  $(v_4, v_0, v_1)$ ; the resulted arc (Figure 3(d)) is  $(v_3, v_0)$ , whose associated dipath in  $\varphi(G)$  is  $(\varphi(v_4), \varphi(v_0), \varphi(v_1))$ , which is, omitting self loops,  $(v_3, v_0)$ . Here  $P_\varphi(e')$  identifies with  $e$ .)

Next, we define two (different) arcs in a directed graph,  $G = (V, E)$ ,  $e_1 = (v_1, u_1)$  and  $e_2 = (v_2, u_2)$  as *parallel*, if, considering  $G$  as undirected (i.e., ignoring the directions of  $E$ ), there is a path between  $v_1$  and  $v_2$  that does not pass through  $u_1$  or  $u_2$ , and there is a path between  $u_1$  and  $u_2$  that does not pass through  $v_1$  or  $v_2$ . The path connecting  $v_1$  and  $v_2$  is allowed to be empty, in case  $v_1 = v_2$ , and analogously for  $u_1$  and  $u_2$ . (See Figure 4)

A contraction  $\varphi$  of  $G$  to  $V'$  is called *condensing* if there exist two parallel arcs in  $G$ ,  $(v_1, u_1)$  and  $(v_2, u_2)$ , such that  $\varphi(v_1) = \varphi(v_2) \neq \varphi(u_1) = \varphi(u_2)$ . Otherwise,  $\varphi$  is said to be *non-condensing*.



**Fig. 4.** Parallel arc pairs are  $(x, u)$  and  $(x, y)$ ,  $(x, u)$  and  $(y, v)$ ,  $(x, u)$  and  $(v, u)$ .

**Lemma 4.** Let  $(H, P)$  be a  $c$ -DVPL of the graph  $G = (V, E)$ . Let  $V'$  be a subset of  $V$ , and let  $\varphi$  be a non-condensing contraction of  $G$  to  $V'$ . Then  $(H_\varphi, P_\varphi)$  is a  $c$ -DVPL of  $\varphi(G)$ .

Now we define a special kind of graph families: Let  $\mathcal{M} = \{G_n\}$  be a family of  $n$ -vertex directed graphs. The family  $\mathcal{M}$  is called *non-condensingly contractable (NCC)* if there exists a family of  $n$ -vertex directed graphs  $\mathcal{M}' = \{F_n\}$ , functions  $g(n, c)$  and  $f(n, c)$  and an integer constant  $c \geq 1$  such that the following properties hold:

- (P1)  $\tilde{\mathcal{H}}_{\max}(G_n, c) \leq g(n, c)$ .
- (P2)  $\tilde{\mathcal{H}}_{\max}(F_n, c) \geq f(n, c)$ .
- (P3) For every constant  $0 < \alpha < 1$  and every  $c \in N$ , there is a function  $p(\alpha, c)$  s.t.  
 $f(\alpha n, c) \geq p(\alpha, c) \cdot g(n, c)$  for every  $n$ .
- (P4) For every graph  $G_n$  and for every subset  $V'$  of  $V$ , there exists a non-condensing contraction  $\varphi$  such that  $\varphi(G_n) \in \mathcal{M}'$ .

Intuitively, an *NCC* graph family is a family whose every graph has “many” pairs of nodes whose mutual distance is “close to” the graph’s diameter, or, in other words, the diameter is “approximately” achieved by “many” pairs of nodes.

For such graph families, we can state and prove our main theorem, which is useful for deriving lower bounds on the total hop count of some graph families.

**Theorem 1.** Let  $\{G_n\}$  be an *NCC* graph family. Then  $\tilde{\mathcal{H}}_{\text{tot}}(G_n, c) = \Theta(n^2 \cdot \tilde{\mathcal{H}}_{\max}(G_n, c))$ .

*Proof.* Consider an *NCC* graph family  $\mathcal{M} = G_n$ . By Lemma 3,  $\tilde{\mathcal{H}}_{\text{tot}}(G_n, c) = O(n^2 \cdot \tilde{\mathcal{H}}_{\max}(G_n, c))$ , so it remains to prove the opposite direction, i.e., to show that the virtualization  $H_n$  that achieves  $\mathcal{H}_{\text{tot}}(H_n) = \tilde{\mathcal{H}}_{\text{tot}}(G_n, c)$  satisfies  $\mathcal{H}_{\text{tot}}(H_n) = \Omega(n^2 \cdot \tilde{\mathcal{H}}_{\max}(G_n, c))$ .

Let  $c_1, c_2$  be constants such that  $0 < c_2 < c_1 < \frac{1}{2}$ . Define  $\alpha = \frac{c_1 - c_2}{1 - c_2}$ , and for each  $c$  define  $\bar{c} = \min\{1, \frac{p(\alpha, c)}{3}\}$ , where  $p(\alpha, c)$  is the function specified for  $G_n$  in property (P3).



Let  $H_n$  be the virtualization of  $G_n$  attaining  $\mathcal{H}_{tot}(H_n) = \tilde{\mathcal{H}}_{tot}(G_n, c)$ . Let  $\delta = \bar{c} \cdot \tilde{\mathcal{H}}_{max}(G_n, c)$ . Let  $x$  denote the number of pairs of nodes  $(u, v)$  in  $V$  whose mutual distance in  $H_n$  satisfies  $d_{H_n}(u, v) \leq \delta$ . If  $x \leq c_1 \cdot n^2$ , then  $H_n$  includes at least  $(\frac{1}{2} - c_1) \cdot n^2 - \frac{n}{2}$  pairs of nodes  $(u, v)$  such that  $d_{H_n}(u, v) > \delta$ , hence

$$\mathcal{H}_{tot}(H_n) \geq \left( \left( \frac{1}{2} - c_1 \right) \cdot n^2 - \frac{n}{2} \right) \cdot \bar{c} \cdot \tilde{\mathcal{H}}_{max}(G_n, c) = \Omega(n^2 \cdot \tilde{\mathcal{H}}_{max}(G_n, c)),$$

and we are done. So hereafter suppose  $x \geq c_1 \cdot n^2$ .

For each node  $v \in H_n$  define the  $\delta$ -neighborhood of  $v$  in  $G(n)$  as

$$N(v) = \{u \in H \mid \min(d_{H_n}(v, u), d_{H_n}(u, v)) \leq \delta\}.$$

We call a node  $u$  *close to*  $v$  if  $u \in N(v)$ , and call a node  $v \in H_n$  a *congestion point* if  $|N(v)| \geq c_2 n$ .

Denote the number of congestion points in  $H_n$  by  $m$ . For a congestion point  $v$ , an upper bound on  $|N(v)|$  is  $n$ , and for a non-congestion point  $v$ ,  $|N(v)| \leq c_2 n$ . So, noting that  $x \leq \sum_{v \in V} |N(v)|$ , we upper bound  $x$  by

$$x \leq mn + c_2 n(n - m).$$

As by assumption  $x$  is no smaller than  $c_1 n^2$ , we get that  $mn + c_2 n(n - m) \geq c_1 n^2$ , which yields

$$m \geq \frac{c_1 - c_2}{1 - c_2} \cdot n = \alpha n.$$

So let us take  $V' \subseteq V$  as an arbitrary set of  $\alpha n$  congestion points. Now we non-condensingly contract  $G_n$  to  $V'$ , denoting by  $F_{\alpha n}$  the resulting graph, and by  $H_{\alpha n}$  the result of  $H_n$  under the contraction. Since the contraction is non-condensing,  $H_{\alpha n}$  is a  $c$ -admissible virtualization of  $F_{\alpha n}$  by Lemma 4. We also have

$$\mathcal{H}_{max}(H_{\alpha n}) \geq f(\alpha n, c) \geq p(\alpha, c) \cdot g(n, c) \geq p(\alpha, c) \cdot \tilde{\mathcal{H}}_{max}(G_n, c).$$

So in  $H_{\alpha n}$  there is a pair of nodes  $(u', v')$  whose distance in  $H_{\alpha n}$  is at least  $d_{H_{\alpha n}}(u', v') \geq p(\alpha, c) \cdot \tilde{\mathcal{H}}_{max}(G_n, c)$ . Their distance in  $H_n$  is definitely no smaller. So we have two *congestion point* nodes  $(u', v')$  in  $H_n$  whose distance is at least  $d_{H_n}(u', v') \geq p(\alpha, c) \cdot \tilde{\mathcal{H}}_{max}(G_n, c)$ . By definition of  $\bar{c}$  we get  $d_{H_n}(u', v') \geq 3\bar{c} \cdot \tilde{\mathcal{H}}_{max}(G_n, c)$ . It follows that  $H_n$  contains at least  $c_2^2 \cdot n^2$  pairs of nodes, namely, the nodes in  $N(u') \times N(v')$ , whose mutual distance is at least  $\bar{c} \cdot \tilde{\mathcal{H}}_{max}(G_n, c)$ . This implies that

$$\tilde{\mathcal{H}}_{tot}(G_n, c) = \mathcal{H}_{tot}(H_n) \geq c_2^2 \cdot n^2 \cdot \bar{c} \cdot \tilde{\mathcal{H}}_{max}(G_n, c) = \Omega(n^2 \cdot \tilde{\mathcal{H}}_{max}(G_n, c)). \quad \blacksquare$$

By establishing that paths, cycles and trees are NCC families, we get

- Corollary 1.** 1. For the  $n$ -vertex path  $P_n$ ,  $\tilde{\mathcal{H}}_{tot}(P_n, c) = \Theta(n^2 \cdot \tilde{\mathcal{H}}_{max}(P_n, c))$ .  
 2. For the  $n$ -vertex cycle  $C_n$ ,  $\tilde{\mathcal{H}}_{tot}(C_n, c) = \Theta(n^2 \cdot \tilde{\mathcal{H}}_{max}(C_n, c))$ .  
 3. For any  $n$ -vertex tree  $T_n$ ,  $\tilde{\mathcal{H}}_{tot}(T_n, 2) = \Theta(n^2 \cdot \tilde{\mathcal{H}}_{max}(T_n, 2))$ .

*Proof.* We have to show that each of the above graph families is *NCC*. In each of the three cases, we take  $F_n = G_n$ . For the  $n$ -vertex path  $P_n$ , as by [6]

$$\frac{n^{\frac{1}{2c-1}}}{2} \leq \tilde{\mathcal{H}}_{max}(P_n, c) \leq 4c \left( \frac{n-1}{2} \right)^{\frac{1}{2c-1}},$$

we can take  $f(n, c) = n^{1/(2c-1)}/2$ ,  $g(n, c) = 4c(\frac{n-1}{2})^{1/(2c-1)}$  and  $p(\alpha, c) = (2\alpha)^{1/(2c-1)}/(8c)$ , so properties (P1)-(P3) hold. As for the last property, (P4), an appropriate contraction of the path is achieved by mapping each non-congestion point that has some congestion points on its left to the closest such point, and mapping each non-congestion point that has no congestion point to its left to the nearest congestion point on its right.

For the  $n$ -vertex cycle  $C_n$ , as by [6]

$$\frac{n^{\frac{1}{2c}}}{2} \leq \tilde{\mathcal{H}}_{max}(C_n, c) \leq 4c \left( \frac{n}{2} \right)^{\frac{1}{2c}},$$

we can take  $f(n, c) = n^{1/(2c)}/2$ ,  $g(n, c) = 4c(\frac{n}{2})^{1/(2c)}$  and  $p(\alpha, c) = (2\alpha)^{1/(2c)}/(8c)$ , satisfying properties (P1)-(P3). An appropriate contraction of the cycle satisfying property (P4) is mapping each non-congestion point to its closest congestion point in the clockwise direction.

For the  $n$ -vertex tree  $T_n$ , as by [6]

$$\frac{1}{2} \cdot n^{1/3} \leq \tilde{\mathcal{H}}_{max}(T_n, 2) \leq 32 \cdot n^{1/3},$$

we take  $f(n, 2) = n^{1/3}/2$ ,  $g(n, 2) = 32 \cdot n^{1/3}$  and  $p(\alpha, 2) = \alpha^{1/3}/64$ , satisfying properties (P1)-(P3), and for property (P4), we contract the tree by taking some congestion point as the root of the tree and mapping each non-congestion point to the closest congestion point on the (unique) path connecting it to the root.

In all three cases we have *NCC* families, so the result follows. ■

Finally, combining Corollary 1 with the bounds established in [6] for  $\tilde{\mathcal{H}}_{max}$  over paths, cycles and trees (as quoted in the proof of the above corollary), we get the following.

**Corollary 2.** 1. For the  $n$ -vertex path,  $P_n$ ,  $\tilde{\mathcal{H}}_{tot}(P_n, c) = \Theta(n^{2+\frac{1}{2c-1}})$ .  
 2. For the  $n$ -vertex cycle,  $C_n$ ,  $\tilde{\mathcal{H}}_{tot}(C_n, c) = \Theta(n^{2+\frac{1}{2c}})$ .  
 3. For an  $n$ -vertex tree,  $T_n$ ,  $\tilde{\mathcal{H}}_{tot}(T_n, 2) = \Theta(n^{\frac{7}{3}})$ .

## 4 Arbitrary Root-to-All Communication on Trees

So far, we concentrated on the DVPL problem over the all-to-all pattern of communication demands, and gave global (*combinatorial*) bounds for the total hop count of admissible (capacity-restricted) virtual graphs. In this section we turn to *algorithmic* aspects, and consider the problem of *finding* a virtual  $c$ -admissible graph which minimizes the total hop count, as well as calculating

that minimal total hop count, for a given pair  $(G, c)$ . This problem seems to be complicated in general, so following [14] we focus on studying it in a more restricted setting. Specifically, we give an algorithm for handling the (*weighted*) *one-to-all* version of the DVPL problem on *trees*.

Formally, we are given an  $n$ -vertex (directed or undirected) tree  $G = (V, E)$  on vertices  $v_1, \dots, v_n$ , rooted at  $v_1$ , with all arcs directed away from the root. The capacity of each edge is specified by a capacity function  $c$ . It is assumed that the capacities are bounded by some constant  $c_0 \in \mathbb{N}$ . (Alternatively, larger capacities can be allowed, so long as the number of different choices for link capacity remains constant; a nonconstant  $c_0$  would make the complexity of our algorithm exponential.) We are also given a *requirements vector*  $\omega$  specifying, for every  $1 < i \leq n$ , the expected amount  $\omega_i$  of traffic between  $v_1$  and  $v_i$ . We call such a pair  $(G, \omega)$  of a graph and a requirements vector a *requirement pair*.

For the pair  $(G, \omega)$ , the  $\omega$ -total hop count is defined by taking into account the relevant communication requirement of each pair of vertices, namely,

$$\mathcal{H}_{tot}^\omega(G) = \sum_{2 \leq i \leq n} d_G(v_1, v_i) \cdot \omega_i.$$

Define  $\tilde{\mathcal{H}}_{tot}^\omega(G, c)$  as the minimum  $\omega$ -total hop count that can be achieved by a  $c$ -admissible virtualization of  $G$ . We want to calculate  $\tilde{\mathcal{H}}_{tot}^\omega(G, c)$ , and find a  $c$ -admissible virtualization of  $G$  that achieves it.

In order to simplify the presentation, we handle first the special case where the requirements in  $\omega$  are boolean, namely,  $\omega_i \in \{0, 1\}$  for every  $1 < i \leq n$ .

For the boolean variant of the problem, the vertices with which the root needs to communicate are called the *required destinations*, and are denoted by  $Q(\omega) = \{v_i \mid \omega_{1i} = 1\}$ .

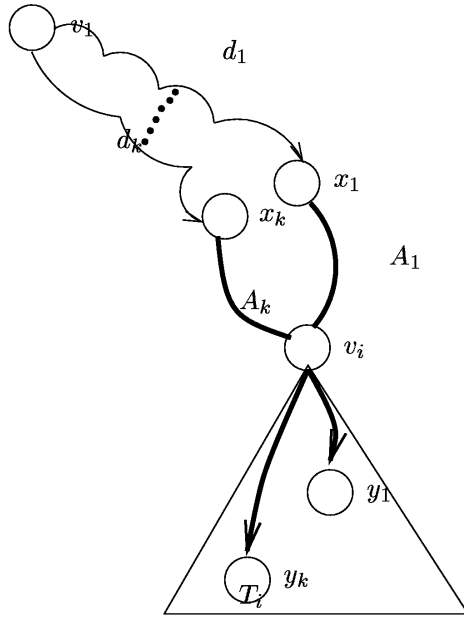
It turns out that this restricted variant of the problem can be solved in polynomial time. For a given tree  $G$  and a set  $Q(\omega)$  of required destinations, the best virtualization is found by dynamic programming. For this, we would like to evaluate the contribution of a subtree to the total hop count. Let the vertices of  $G$  other than the root be  $v_2, \dots, v_n$ , organized in breadth-first order, i.e., satisfying that if  $v_i$  is the parent of  $v_j$ , then  $i < j$ . Each  $v_i$  defines a subtree of  $G$ , denoted  $T_i$ , whose set of vertices  $V_i$  includes  $v_i$  and all the vertices below it in the tree. We also denote by  $e_i$  the arc entering  $v_i$ . For some virtualization  $H$  of  $G$ , and some subtree  $T_i$  of  $G$ , we define the *internal total hop count of  $T_i$  with respect to  $H$*  as

$$\mathcal{H}_{tot}^\omega(H, T_i) = \sum_{v_j \in V_i \cap Q(\omega)} d_H(v_1, v_j).$$

We also define the *minimal required total hop count of  $T_i$*  as the minimum over all such virtualizations as

$$\tilde{\mathcal{H}}_{tot}^\omega(T_i) = \min\{\mathcal{H}_{tot}^\omega(H, T_i) \mid H \in \text{Virt}(G)\}.$$

As usual, the dynamic programming algorithm is based on solving many small subproblems gradually. A typical subproblem  $\Psi(T_i, \mathbf{d})$  to be solved during the algorithm is defined as follows:



**Fig. 5.** A typical subproblem to be solved.

**Input:** A subtree  $T_i$  rooted at  $v_i$  and a tuple  $\mathbf{d} = \langle d_1, \dots, d_k \rangle$  of  $k$  non-negative integers.

Intuitively, the tuple represents the starting points of  $k \geq 0$  virtual arcs  $\mathcal{A} = \langle A_1, \dots, A_k \rangle$  entering  $T_i$ . Each arc  $A_j$  starts at some node  $x_j$  on the path between  $v_1$  and  $v_i$ . The input component  $d_i$  represents the distance in  $H$  of its start vertex  $x_j$  from  $v_1$ , namely  $d_j = d_H(v_1, x_j)$ . This value is henceforth referred to as the *tail-length* of  $A_j$ . The end vertex of each of the  $k$  arcs is some (unknown) vertex  $y_j$  of  $T_i$  (see Figure 5). Hence these virtual arcs are not yet completely specified, as their end-points  $y_j$  are to be chosen by the solution devised for the subproblem  $\Psi$ . It is convenient to refer to the virtual arcs of  $\mathcal{A}$  throughout the following discussion, but the reader should bear in mind that these arcs are only implicit in the algorithm, and only the  $\mathbf{d}$  vectors are manipulated explicitly.

**Output:** A value  $f(T_i, \mathbf{d})$ , which is a non-negative integer or  $\infty$ .

For a subtree  $T_i$  rooted at  $v_i$  and a tuple  $Y = \langle y_1, \dots, y_k \rangle$  of end vertices of the virtual arcs, we denote by  $H_Y$  the virtualization of  $T_i$  including all *internal* virtual arcs (i.e., arcs of  $H$  downwards from vertices of  $T_i$ ), and the  $k$  paths from  $v_1$  to each  $y_j$  of lengths  $d_j + 1$ , for  $1 \leq j \leq k$ . Let  $Y^*$  denote the optimal tuple  $Y$ , such that the internal total hop count  $\mathcal{H}_{tot}^\omega(H_{Y^*}, T_i)$  of  $T_i$  is minimal over all virtualizations including  $k$  paths from  $v_1$  to some vertices in  $T_i$  of lengths  $d_j + 1$ . The output of  $\Psi(T_i, \mathbf{d})$ , denoted by  $f(T_i, \mathbf{d})$ , is  $\mathcal{H}_{tot}^\omega(H_{Y^*}, T_i)$  if this can be done, or  $\infty$  if the instance  $(T_i, \mathbf{d})$  is infeasible.

We calculate  $\tilde{\mathcal{H}}_{tot}^\omega(G, c)$  (and find the virtualization that achieves it), by solving all the subproblems  $\Psi(T_i, \mathbf{d})$ , namely, finding  $f(T_i, \mathbf{d})$  for every  $T_i$  and every possible tuple of tail-lengths  $\mathbf{d}$ . This is done according to the dynamic programming paradigm. We create a table  $F$  of  $n - 1$  rows and  $\frac{n^{c_0+1}-1}{n-1}$  columns. The columns are grouped into sets  $F_0, F_1, \dots, F_{c_0}$ , where  $F_k$  consists of  $n^k$  columns. These columns are used to store all possible outputs  $f(T_i, \mathbf{d})$ . Row  $i$  of the table stands for the subtree  $T_i$  rooted at the vertex  $v_i$ . Note that the rows are ordered  $0, 1, \dots, n - 1$ , so if  $v_i$  is a parent of  $v_j$ , then  $T_j$ 's row is above  $T_i$ 's row. The entries of this set of columns and of the  $i$ th row stand for the class of subproblems in which  $k$  virtual arcs enter  $T_i$ .

As  $d_H(v_1, x_j) \leq n - 1$ , for every  $x_j \in V$  and every virtualization  $H$  including a path from  $v_1$  to  $x_j$ , the  $k$ th set has  $n^k$  columns, each standing for one tuple of  $\mathbf{d}$ . The procedure will store  $f(T_i, \mathbf{d})$  in the entry of the  $i$ th row and the column corresponding to  $\mathbf{d}$ .

Clearly, if  $k = 0$ , and  $T_i$  includes any required destinations, then the subproblem  $\Psi(T_i, \mathbf{d})$  for  $\mathbf{d} = \langle \rangle$  is infeasible, so the value for this entry is  $\infty$ . Note that if the (physical) arc reaching  $v_i$  is  $e_i$ , then the subproblem  $\Psi(T_i, \mathbf{d})$  for  $|\mathbf{d}| > c(e_i)$  is also infeasible, so the columns in sets  $F_k$  for  $k > c(e_i)$  (if there are any) are irrelevant, and should also be filled with  $\infty$ . Note also that for each  $T_i$  there is no need to use more virtual arcs than the number of required destinations in  $T_i$ , but we do take such cases into consideration during the table filling.

Note that if  $k \geq 1$  arcs reach  $v_i$ , then it is never necessary to terminate more than one of them at  $v_i$  itself. Hence we can extend some (perhaps all) of them to reach some internal vertices of  $T_i$ . The way we do it affects the needed internal total hop count.

The algorithm fills the table in a dynamic programming manner, from the bottom up, filling each row by using the already filled ones below it. In the full paper we present the algorithm in detail, prove that it indeed calculates  $\tilde{\mathcal{H}}_{tot}^\omega(G, c)$ , and bound its complexity, yielding the following theorem.

**Theorem 2.** *The algorithm described above calculates  $\tilde{\mathcal{H}}_{tot}^\omega(G, c)$ , and finds the virtualization that achieves it in polynomial time.*

The algorithm as presented assumes that the requirements vector  $\omega$  is boolean. We now describe how this algorithm can be extended to the general case.

The notation and the algorithm are a natural generalization of those of the previous section, with some minor changes. Having a general communication requirements vector  $\omega$  for the tree graph  $G$ , for some virtualization  $H$  of  $G$ , and some subtree  $T_i$  the *internal total hop count of  $T_i$  with respect to  $H$*  is defined with respect to  $\omega$ :

$$\mathcal{H}_{tot}^\omega(H, T_i) = \sum_{v_j \in V_i} d_H(v_1, v_j) \cdot \omega_j,$$

and the definition of  $\tilde{\mathcal{H}}_{tot}^\omega(T_i)$  is modified accordingly. The dynamic paradigm calculations and the table filling are done as in the previous section, only filling leaf values with respect to  $\omega$ :

$$f(T_i, \mathbf{d}) = \begin{cases} 0, & k = 0 \text{ and } \omega_{1i} = 0, \\ \infty, & k = 0 \text{ and } \omega_{1i} \neq 0, \\ \min_{1 \leq j \leq k} \{(d_j + 1) \cdot \omega_{1i}\}, & 0 < k \leq c(e_i), \\ \infty, & k > c(e_i). \end{cases}$$

Also taking the communication requirements into consideration when calculating the function  $g$ :

$$g(T_i, \mathbf{d}, X) = (d_0 + 1) \cdot \omega_{1i} + \sum_{j=1}^m f(T_{i_j}, \mathbf{d}^j(\hat{X})).$$

The correctness proof and complexity analysis of the algorithm remain the same, and so does its modification, which enables us *finding* the virtualization that achieves it. Therefore, we have the following.

**Theorem 3.** *The algorithm described above calculates  $\tilde{\mathcal{H}}_{tot}^\omega(G, c)$ , and finds the virtualization that achieves it in polynomial time.*

## References

1. B. Awerbuch, A. Bar-Noy, N. Linial and D. Peleg, Compact distributed data structures for adaptive routing, *Proc. 21st ACM Symp. on Theory of Computing*, Seattle, Washington, pp. 479-489, May 1989.
2. B. Awerbuch and D. Peleg, Routing with polynomial communication-space trade-off, *SIAM Journal on Discrete Mathematics*, Vol. 5, No. 2, pp. 151-162, 1992.
3. S. Ahn, R.P. Tsang, S.R. Tong and D.H.C. Du, Virtual path layout design on ATM networks, *Proc. IEEE INFOCOM'94*, pp. 192-200, 1994.
4. L. Becchetti, P. Bertolazzi, C. Gaibisso and G. Gambosi, On the design of efficient ATM routing schemes, manuscript 1997.
5. J. Burgin and D. Dorman, Broadband ISDN resource management: The role of virtual paths, *IEEE Communications Magazine*, 29, 1991.
6. J.C. Bermond, N. Marlin, D. Peleg and S. Perennes, Directed Virtual Path Layouts in ATM networks, *Proc. 12th Int. Symp. on Distributed Computing*, , Andros, Greece, pp. 75-88, Oct. 1998.
7. P. Chanas and O. Goldschmidt, Conception de réseau de VP de diamètre minimum pour les réseaux ATM, *Road-f'98*, pp. 38-40, 1998.
8. I. Cidon, O. Gerstel and S. Zaks, A scalable approach to routing in ATM networks, *Proc. 8th Int. Workshop on Distributed Algorithms (WDAG)*, LNCS 857, Springer Verlag, Terschelling, The Netherlands, pp. 209-222, October 1994.
9. R. Cohen and A. Segall, Connection management and rerouting in ATM networks, *Proc. IEEE INFOCOM'94*, pp. 184-191, 1994.
10. T. Eilam, M. Flammini and S. Zaks, A complete characterization of the path layout construction problem for ATM networks with given hop count and load, *Proc. 24th Int. Colloq. on Automata, Languages and Programming (ICALP)*, LNCS 1256, Springer-Verlag, pp. 527-537, 1997.
11. M. Feiglstein and S. Zaks, Duality in chain ATM virtual path layouts, *Proc. 4th Int. Colloq. on Structural Information and Communication Complexity (SIROCCO)*, Monte Verita, Ascona, Switzerland, July 1997.

12. O. Gerstel, *The Virtual Path Design in ATM Networks*, PhD thesis, Department of Computer Science, Technion, Haifa, Israel, December 1995.
13. O. Gerstel, I. Cidon and S. Zaks, The layout of virtual path in ATM networks, *IEEE Transactions on Networking*, 4(6):873-884, 1996.
14. O. Gerstel, A. Wool and S. Zaks, Optimal layouts on a chain ATM network, *Discrete Applied Mathematics*, 83:157-178, 1998.
15. O. Gerstel and S. Zaks, The virtual path layout problem in fast networks, *Proc. 13th ACM Symp. on Principles of Distributed Computing (PODC)*, Los Angeles, CA, U.S.A., pp. 235-243, August 1994.
16. R. Händler and M.N. Huber, *Integrated Broadband Networks: an introduction to ATM-based networks*, Addison-Wesley, 1991.
17. ITU recommendation, I series (B-ISDN), Blue Book, November 1990.
18. E. Kranakis, D. Krizanc and A. Pelc, Hop-congestion tradeoffs for high-speed networks, *Int. Journal of Foundations of Computer Science*, 8, 1997.
19. F.Y.S. Lin and K.T. Cheng, Virtual path assignment and virtual circuit routing in ATM networks, *Proc. IEEE GLOBECOM'93*, pp. 436-441, 1993.
20. C. Partridge, *Gigabit Networking*, Addison Wesley, 1994.
21. K.I. Sato, S. Ohta and I. Tokizawa, Broadband ATM network architecture based on virtual paths, *IEEE Transactions on Communications*, Vol. 38, No. 8, pp. 1212-1222, August 1990.
22. Y. Sato and K.I. Sato, Virtual path and link capacity design for ATM networks, *IEEE, Journal of Selected Areas in Communications*, Vol. 9, 1991.
23. S. Zaks, Path Layout in ATM networks- a survey, *The DIMACS Workshop on Networks in Distributed Computing*, DIMACS Center, Rutgers University, October 1997.

# Efficient Routing in Networks with Long Range Contacts

## (Extended Abstract)\*

Lali Barrière<sup>1</sup>, Pierre Fraigniaud<sup>2</sup>, Evangelos Kranakis<sup>3</sup>, and Danny Krizanc<sup>4</sup>

<sup>1</sup> Dept. de Matemàtica Aplicada i Telemàtica, Universitat Politècnica de Catalunya.  
Email: lali@mat.upc.es

<sup>2</sup> CNRS, LRI, Université Paris-Sud, France. Email: pierre@lri.fr

<sup>3</sup> School of Computer Science, Carleton University, Canada. Email: kranakis@scs.carleton.ca  
Research supported in part by NSERC (Natural Sciences and Engineering Research Council of Canada).

<sup>4</sup> Department of Mathematics, Wesleyan University, USA. Email: dkrizanc@caucus.cs.wesleyan.edu

**Abstract.** We investigate the notion of Long Range Contact graphs. Roughly speaking, such a graph is defined by (1) an underlying network topology  $G$ , and (2) one (or possibly more) extra link connecting every node  $u$  to a “long distance” neighbor, called the long range contact of  $u$ . This extra link represents the *a priori* knowledge that a node has about far nodes and is set up randomly according to some probability distributions  $p$ . To illustrate the claim that Long Range Contact graphs are a good model for the small world phenomenon, we study greedy routing in these graphs. Greedy routing is the distributed routing protocol in which a node  $u$  makes use of its long range contact to progress toward a target, if this contact is closer to the target, than the other neighbors. We give upper and lower bounds on greedy routing on the  $n$ -node ring  $C_n$  augmented with links chosen using the  $r$ -harmonic distributions. In particular, we show a tight  $\Theta(\log^2 n)$ -bound for the expected number of steps required for routing in  $C_n$  augmented using the 1-harmonic distribution. Hence, our study shows that the model of Kleinberg [11] can be simplified by using the ring rather than the mesh while preserving the main features of the model. Our study also demonstrates the significant difference (in term of both diameter and routing) between the ring augmented with long range contacts chosen with the harmonic distribution and the ring augmented with a random matching as introduced by Bollobas and Chung [3]. Finally, using epimorphisms of a graph onto another, for any network  $G$ , we show how to define a probability distribution  $p$  and study the performance of greedy routing in  $G$  augmented with  $p$ . For appropriate embeddings (if they exist), this performance turns out to be  $O(\log^2 n)$ .

---

\* Part of this work was done while the first and third author were visiting LRI at Orsay (additional support from Université Paris-Sud), and while the first and second authors were visiting Carleton University. Additional support from Carleton University, NATO, and MITACS (Mathematics of Information Technology and Complex Systems) grants.



# 1 Introduction

The small-world phenomenon arises from rather anecdotal experience that has been witnessed in many large interconnected systems: it is a phenomenon that formalizes the paradoxical ability of an entity in the system to be only a few “degrees” of separation away from any other entity in the system. This paradoxical occurrence of the small-world phenomenon has been backed by statistical data of reachability and has several instantiations in the scientific literature from sociology to the web. It has become the subject of investigation in popular as well as artistic culture (see [78,16]).

To understand this phenomenon studies have been made that include the introduction of two graph theoretic models: *relational* graphs and *spatial* graphs. In relational graphs the probability of the vertices becoming connected depends only upon preexisting connections [35,16,17]. In spatial graphs, the corresponding probability is a function of the vertices [11,16]. In recent years the web has been the focus of investigations. Here researchers have investigated power-laws, i.e., the probability that a node has degree  $k$  is given by  $k^{-c}$ , for some constant  $c > 0$ ; this implies that nodes with low degree are the most numerous and the probability of nodes with given degree  $k$  decreases as  $k$  increases proportionately with  $k^{-c}$  [14,16,12]. All these studies show that random graphs  $\mathcal{G}_{n,p}$  as defined by Erdős and Rényi, are not good models for the small world phenomenon, because they have a large diameter when the average degree is small [2].

In this paper, we study the notion of Long Range Contact graphs. Let  $G = (V, E)$  be a network on  $n$  vertices. Consider a probabilistic mapping  $p$  on the vertices of  $G$  such that  $\sum_{v \in V} p(u, v) = 1$ , for all  $u \in V$ . I.e., each node  $u \in V$  has an associated probability distribution  $p(u, \cdot)$ . Given  $G$  and  $p$ , the Long Range Contact graph  $(G, p)$  is a directed graph defined on the same set of vertices, such that every node  $u$  has  $\deg_G(u) + 1$  out-neighbors, that is its  $\deg_G(u)$  neighbors in  $G$ , plus one additional out-neighbor chosen at random according to  $p$ . This latter neighbor is called the *long range contact* of  $u$ . The probabilistic mapping  $p$ , i.e., the probability distributions  $p(u, \cdot)$ ’s, reflect “vague knowledge” available at the nodes about the possible status and location of a desired information located at some node of the network.

In small world graphs, not only have the nodes a few degrees of separation, but these nodes are able (or expected) to find reasonably short routes between them. Therefore, the following two parameters have been the source of much research: (1) The *diameter* of  $(G, p)$ , i.e., the maximum distance between any two nodes in the augmented graph; and (2) The performance of *greedy routing* in  $(G, p)$ , i.e., routing from a source  $s$  to a target  $t$  is executed by selecting, at each intermediate node  $u$ , the next node as the neighbor of  $u$  (including its long range contact) which is closer (in the graph  $G$ ) to the target  $t$ .

These two parameters depend first on the probability distribution to select a long range contact and second on the underlying topology of the graph. To be a good candidate to abstract small world phenomenon, a graph model must insure

**Table 1.** Expected number of steps of greedy routing in the ring augmented with long range contacts chosen according to the  $r$ -harmonic distribution.

$r$ -Harmonic Distribution	Lower Bound	Reference	Upper Bound	Reference
$0 \leq r < 1$	$\Omega(n^{\frac{1-r}{2-r}})$	Theorem <a href="#">4</a>	$O(n^{1-r})$	Theorem <a href="#">2</a>
$r = 1$	$\Omega(\log^2 n)$	Theorem <a href="#">5</a>	$O(\log^2 n)$	Theorem <a href="#">1</a>
$1 < r < 2$	$\Omega(n^{\frac{r-1}{r}})$	Theorem <a href="#">4</a>	$O(n^{r-1})$	Theorem <a href="#">1</a>
$r = 2$	$\Omega(\sqrt{n})$	Theorem <a href="#">4</a>	$O(\frac{n \log \log n}{\log n})$	Theorem <a href="#">3</a>
$2 < r$	$\Omega(n^{\frac{r-1}{r}})$	Theorem <a href="#">4</a>	$O(n)$	Trivial

that both the diameter, and the number of greedy routing steps, be small. In this paper, we study the model in which  $G$  is the ring  $C_n$ , and  $p$  is the harmonic distribution.

**Related research.** Among the previously cited papers, two are strongly connected to this paper. Bollobas and Chung [3](#) have studied the diameter of a ring plus a random matching, selected uniformly among all possible matchings. They have shown that the resulting augmented ring has a diameter  $\Theta(\log n)$  with a probability tending to 1 as  $n$  goes to infinity. However, the performance of greedy routing can be very bad in this model. Indeed, Kleinberg [11](#) has shown that the ring augmented with long range contacts chosen uniformly at random offers very bad properties in term of routing ( $\Omega(\sqrt{n})$  lower bound for the expected number of steps). As an attempt to model the small world phenomenon, Kleinberg has therefore proposed to use the 2-dimensional square grid augmented with long range contacts chosen according to the 2-harmonic distribution. He showed that, in this model, greedy routing performs in  $O(\log^2 n)$  expected number of steps. Moreover he showed that this is optimal in the sense that for  $r \neq 2$  any distributed routing algorithm based on the  $r$ -harmonic distribution has an  $n^{\Omega(1)}$  lower bound on the expected number of steps. He concluded that the grid with the 2-harmonic distribution is a good model for the small world phenomenon.

**Results of the paper.** Motivated by the research of Bollobas and Chung, we have investigated the augmented ring. Motivated by the research of Kleinberg, we have investigated  $r$ -harmonic mappings  $p_r$ ,  $r \geq 0$ , defined as follows. Given two nodes  $u$  and  $v$ , the probability for  $u$  to have  $v$  as long range contact is given by  $p_r(u, v) = \frac{d(u, v)^{-r}}{\sum_{w \neq u} d(u, w)^{-r}}$ , where  $d(\cdot, \cdot)$  is the distance function in the network. The *uniform* distribution (which is obtained for  $r = 0$ ), i.e.,  $p(i, j) = 1/n$ , and the *Zipf* distribution [18](#) (which is obtained for  $r = 1 - \log .80 / \log .20$ ), are two examples of harmonic distributions. We have performed an exhaustive study of the performances of greedy routing in the ring augmented with harmonic long range contacts, for all  $r \geq 0$ . Table [1](#) summarizes our results.

One important result in this table is the tight  $\Theta(\log^2 n)$ -bound for the expected number of steps of greedy routing in the ring augmented with long range contact chosen using the 1-harmonic distribution. The upper bound  $O(\log^2 n)$  shows that the simple ring can perform as well as the square mesh, and hence provides a simpler model for the small world phenomenon. The lower bound  $\Omega(\log^2 n)$ , as well as the other lower bounds for  $r \neq 1$ , show that greedy routing cannot perform faster than  $\log^2 n$  steps in any ring augmented with an harmonic distribution. It seems to be a challenging task to prove or disprove the existence of a distribution allowing greedy routing to perform faster in the ring, the square grid, or even the  $k$ -dimensional mesh,  $k \geq 3$ .

As a last contribution, we show how to extend the results of the ring to any network  $G$ , by using epimorphisms of a graph onto another. In particular, we show how to define a probabilistic mapping  $p$  and study the performance of greedy routing in  $(G, p)$ . For appropriate embeddings this performance turns out to be  $O(\log^2 n)$ .

## 2 Preliminary Results

For the purpose of simplification of the presentation, all our results are formally proven for the *directed* ring, i.e., the digraph in which nodes are labeled from 0 to  $n$ , and where node  $i$  has node  $i + 1$  as out-neighbor, and  $i - 1$  as in-neighbor (unless specified otherwise, all operations are performed modulo  $n + 1$ ). In each case, the result in the undirected ring differs by a constant factor only. We denote by  $R_{n+1}$  the directed ring of  $n + 1$  nodes.

The  $r$ -harmonic random variable  $H_r$ , with values in  $\{1, \dots, n\}$  has the probability distribution defined by  $\Pr(\{H_r = k\}) = \frac{k^{-r}}{H_n^{(r)}}$ , where  $H_n^{(r)} = \sum_{i=1}^n i^{-r}$  is the  $r$ -harmonic number of order  $n$ . Therefore, if  $R_{n+1}$  is augmented using the  $r$ -harmonic mapping  $p_r$ , then, given two nodes  $i$  and  $j$ , the probability for  $i$  to have  $j$  as long range contact in  $(R_{n+1}, p_r)$  is given by  $p_r(i, j) = \frac{((j-i) \bmod n+1)^{-r}}{H_n^{(r)}}$ . This formula can be made more explicit by noticing that the harmonic numbers satisfy the following identities.

**Lemma 1.** *The  $r$ -harmonic number of order  $n$  is*

$$H_n^{(r)} = \begin{cases} \frac{1}{1-r} n^{1-r} + O(1) & \text{if } r < 1; \\ \log n + O(1) & \text{if } r = 1; \\ O(1) & \text{if } r > 1. \end{cases}$$

The next lemma shows thresholds in the behavior of the harmonic distributions. Not surprisingly, these thresholds are those appearing in Table [1](#).

**Lemma 2.** *The expected value of  $H_r$  is*

$$\mathbf{E}(H_r) = \begin{cases} \Theta(n) & \text{if } 0 \leq r < 1 \\ \Theta(n/\log n) & \text{if } r = 1 \\ \Theta(n^{r-1}) & \text{if } 1 < r < 2 \\ \Theta(1/\log n) & \text{if } r = 2 \\ \Theta(1) & \text{if } 2 < r. \end{cases}$$

For our analysis of greedy routing in  $R_{n+1}$ , we will always assume that the source node is 0, and the target node is  $n$ . It is indeed easy to observe that this is a worst case, as far as greedy routing is concerned. Indeed, the probability for a node to have a long range contact at distance  $d$  on the ring decreases as  $d$  increases. Therefore the farther a source is from a target, the larger is the expected number of steps to route from that source to that target.

A very naive interpretation of Lemma 2 would be to derive that, e.g., greedy routing in the ring augmented with the 1-harmonic distribution performs in  $O(\log n)$  expected number of steps. This reasoning fails because the expected gain of using long range contacts decreases as one gets closer to the destination (as long range contacts may lead farther away from that destination than one currently is). The following clarifies that point. Given a node  $s \in \{0, \dots, n-1\}$ , greedy routing defines a random variable  $J_s$  as the length of the “jump” performed at  $s$  toward the target  $n$ . It satisfies:  $J_s = H_r$  if  $H_r \leq n-s$ , and  $-1$  otherwise. One can easily show the following.

**Lemma 3.** *For  $k \leq n-s$ , we have*

$$\Pr(\{J_s = k\}) = \begin{cases} \Pr(\{H_r = k\}) + \Pr(\{H_r > n-s\}) & \text{if } k = 1 \\ \Pr(\{H_r = k\}) & \text{if } 1 < k \leq n-s. \end{cases}$$

*And the expected value of the jump  $J_s$  at node  $s$  in  $(R_{n+1}, p_r)$  is:*

$$\mathbf{E}(J_s) = \begin{cases} \Theta((n-s)^{2-r}/n^{1-r}) & \text{if } r < 1 \\ \Theta((n-s)/\log n) & \text{if } r = 1 \\ \Theta((n-s)^{2-r}) & \text{if } 1 < r < 2 \\ \Theta(\log(n-s)) & \text{if } r = 2 \\ \Theta(1) & \text{if } r > 2. \end{cases}$$

### 3 Upper Bounds

We begin with general considerations which apply to arbitrary networks. Then we will refine these concepts for the specific case of the ring. For each vertex  $u$  of  $G = (V, E)$ , and each real number  $r > 0$ , define the ball  $B_r^G(u)$  of radius  $r$  around  $u$  as the set of vertices at distance at most  $r$  from  $u$ . (If the graph used is clear from the context we will omit the superscript  $G$  from  $B_r^G(u)$  and write  $B_r(u)$ .) For any set  $S$  of vertices of  $(G, p)$  and any vertex  $u \in V$  define  $p[u \rightarrow S] = \sum_{v \in S} p(u, v)$ . Here we are trying to quantify the weight that a node  $u$  gives to a contact in  $S$  in the sense that  $p[u \rightarrow S]$  is the probability that a node  $u$  has a long-range contact in the set  $S$ .

**Definition 1.** Let  $G$  be a graph,  $p$  a probabilistic mapping on  $G$ ,  $c > 1$  a constant, and  $f$  a function. The pair  $(G, p)$  is called an  $(f, c)$ -Long Range Contact graph if for any pair  $(u, t)$  of vertices of  $G$  at distance at most  $d$  we have that  $p[u \rightarrow B_{d/c}(t)] \geq \frac{1}{f(d)}$ .

**Lemma 4.** Let  $G = (V, E)$  be a graph of diameter  $D$ . If  $(G, p)$  is an  $(f, c)$ -Long Range Contact graph then greedy routing in  $(G, p)$  performs in  $O\left(\sum_{i=1}^{\log_c D} f(D/c^i)\right)$  expected number of steps.

*Proof.* What is the probability, for a given node  $u$  at distance at most  $d$  from the target  $t$ , that the long range contact selected is at a distance at most  $d/c$  from the target? By definition, this is equal to  $p[u \rightarrow B_{d/c}(t)]$ . Moreover, by the geometric distribution, the expected number of trials to guarantee success is  $1/p[u \rightarrow B_{d/c}(t)]$ . When a trial fails, we make a move towards the target by going to a neighbor along a shortest path from the current node to the target. The next trial is therefore performed at a node still at distance at most  $d$  from  $t$ . It follows from Definition 1 that the expected number of trials to get a contact in  $B_{d/c}(t)$  is at most  $\frac{1}{p[u \rightarrow B_{d/c}(t)]} \leq f(d)$ . This implies that after at most  $f(d)$  expected number of routing steps from  $u$ , we enter  $B_{d/c}(t)$ . Iterating this we conclude that the expected number of steps for routing is at most  $O\left(\sum_{i=1}^{\log_c D} f(D/c^i)\right)$ .

Using specific probabilistic mappings we can simplify our analysis.

**Definition 2.** A probabilistic mapping  $p$  on a graph  $G$  is distance-invariant if  $p(u, v)$  depends only on the distance  $d(u, v)$ . A distance-invariant mapping is called non-increasing if it is a non-increasing function of the distance.

To simplify notation we use the same symbol to denote the resulting mapping, namely  $p(u, v) = p(d(u, v))$ . We can prove the following result.

**Lemma 5.** If  $p$  is a non-increasing distant-invariant mapping on the graph  $G$  then for all vertices  $u, t$  with  $d(u, t) \leq d$  and all constants  $c > 0$ , we have that  $p[u \rightarrow B_{d/c}(t)] \geq p((c+1)d/c) \cdot |B_{d/c}(t)|$ .

*Proof.* Let  $v$  be a node in  $B_{d/c}(t)$ . For any node  $u$ ,  $d(u, v) \leq d(u, t) + d(t, v) \leq d + d/c = (c+1)d/c$ . It follows that

$$\begin{aligned} p[u \rightarrow B_{d/c}(t)] &= \sum_{v \in B_{d/c}(t)} p(u, v) \\ &= \sum_{v \in B_{d/c}(t)} p(d(u, v)) \text{ since } p \text{ is distance invariant} \\ &\geq \sum_{v \in B_{d/c}(t)} p((c+1)d/c) \text{ since } p \text{ is non increasing} \\ &= p((c+1)d/c) \cdot |B_{d/c}(t)|, \end{aligned}$$

which completes the proof of the lemma.

As a direct consequence of Lemma 5 and by definition of  $(f, c)$ -Long Range Contact graphs, we obtain the following result.

**Lemma 6.** *Consider a graph  $G$  and a non-increasing distance-invariant mapping  $p$ . Then, for any  $c > 1$ , the pair  $(G, p)$  is an  $(f, c)$ -Long Range Contact graph where the function  $f(d)$  is defined by  $f(d) = \frac{1}{p((c+1)d/c) \cdot \min_{t \in V} |B_{d/c}(t)|}$ .*

**Theorem 1.** *The expected number of steps for greedy routing on  $R_{n+1}$  is*

$$\begin{cases} O(\log^2 n) & \text{if } r = 1 \\ O(n^{r-1}) & \text{if } 1 < r < 2. \end{cases}$$

*Proof.* The  $r$ -harmonic mapping  $p_r$  on a graph  $G$  is a non-increasing distance-invariant mapping. From Lemma 6  $(G, p_1)$  is a Long Range Contact graph with  $f(d) \simeq 1/\log n$ . The  $O(\log^2 n)$  bound then results from the application of Lemma 4. Similarly, for  $r > 1$ ,  $G(p_r)$  is a Long Range Contact graph with  $f(d) \simeq d^{r-1}$  (cf. Lemma 1). The result then follows by application of Lemma 4.

In order to obtain non trivial upper bounds when either  $r < 1$  or  $r \geq 2$  we can use the method of probabilistic recurrences. First we recall the following discussion from [14] (Theorem 1.3, page 15). Let  $g(x)$  be a monotone non-decreasing function from positive reals to positive reals. Consider a particle starting from position 0 and moving along the discrete line segment from 0 to  $n$  and whose position changes in discrete time intervals. If the particle is currently at position  $s$  it moves to position  $s + X$  where  $X$  is a random variable ranging over the integers  $1, \dots, n - s$  such that  $\mathbf{E}[X] \geq g(n - s)$ . The following result due to Karp, Upfal and Widgerson was first stated in [10] (see also [9] for additional information on probabilistic recurrences):

**Lemma 7.** (Karp, Upfal, Widgerson [10]) *Let  $T$  be the random variable denoting the number of steps in which the particle reaches the position  $n$ . Then  $\mathbf{E}(T) \leq \int_1^n dx/g(x)$ .*

We can use Lemma 7 to analyze greedy routing when  $r < 1$ . More precisely, we can prove the following result.

**Theorem 2.** *The expected number of steps for greedy routing on  $R_{n+1}$  using  $r$ -harmonic distributions with  $0 \leq r < 1$  is  $O(n^{1-r})$ .*

*Proof.* Greedy routing is similar to the motion of the particle described above. By Lemma 3 if the particle is in position  $s$  then the expected length of a jump is  $\Theta((n - s)^{2-r}/n^{1-r})$ . If we let  $g(x) = \Theta(x^{2-r}/n^{1-r})$  then Lemma 7 is applicable and we obtain that the expected number of steps of greedy routing is at most

$$\int_1^n \frac{dx}{x^{2-r}/n^{1-r}} = \frac{n^{1-r}}{1-r} - \frac{1}{1-r}.$$

The Lemma on probabilistic recurrences can also be used for analysing greedy routing when using 2-harmonic distributions.

**Theorem 3.** *The expected number of steps for greedy routing on  $R_{n+1}$  using 2-harmonic distribution is  $O(\frac{n \log \log n}{\log n})$ .*

*Proof.* Combining Lemmas 3 and 7, we can show that up to a constant the expected number of steps of greedy routing is at most  $\int_2^n \frac{dx}{\log x}$ . This is easily seen to be in  $O(\frac{n \log \log n}{\log n})$ .

## 4 Lower Bounds

The proof of the following result is based on a proof in [11].

**Lemma 8.** *Let  $p$  be any distance-invariant mapping on  $R_{n+1}$ . Assume that there exists  $d$  and  $D$ , and  $\epsilon$ ,  $0 < \epsilon < 1$ , such that one of the two following conditions holds:*

1.  $d \geq D$  and  $D \cdot \sum_{i=1}^d p(i) \leq \epsilon$ ;
2.  $d \cdot D < n$  and  $D \cdot \sum_{i>d} p(i) \leq \epsilon$ .

*Then the expected number of steps of greedy routing is at least  $(1 - \epsilon)D$ .*

*Proof.* First we prove the lemma under condition 1. Let  $B$  denote the ball of  $R_{n+1}$  centered at  $n$  and radius  $d$ , i.e.,  $B = \{n - d, \dots, n - 1, n\}$ . Recall that we consider greedy routing from 0 to  $n$ . Consider the events:

- $E$ : In at most  $D$  steps we reach  $n$ .
- $E'$ : In at most  $D$  steps we reach a node that has a long range contact to a node in  $B$ .
- $E'_i$ : In step  $i$  we reach a node that has a long range contact to a node in  $B$ .

Let  $X$  be the random variable which counts the number of steps to reach  $n$  from 0. In view of condition 1 we have that

$$\Pr(E') = \Pr(\cup_{i=1}^D E'_i) \leq \sum_{i=1}^D \Pr(E'_i) \leq D \cdot p[0 \rightarrow B] \leq D \cdot \sum_{i=1}^d p(i) \leq \epsilon.$$

It follows that

$$\Pr(\overline{E'}) = 1 - \Pr(E') \geq 1 - \epsilon. \quad (1)$$

Since  $d \geq D$ ,  $E \subseteq E'$ , and hence  $\overline{E'} \subseteq \overline{E}$ . It follows that  $\Pr(E|\overline{E'}) = 0$ . Using this and Inequality 1 we can show that

$$\mathbf{E}[X] = \sum_k k \cdot \Pr(\{X = k\}) \geq \sum_k k \cdot \Pr(\{X = k\} \cap \overline{E'}) = \Pr(\overline{E'}) \cdot \mathbf{E}[X|\overline{E'}] \geq (1 - \epsilon)D.$$

This proves the first part of the lemma. Next we prove the lemma under condition 2. Consider the events

- $E$ : In at most  $D$  steps we reach  $n$ .
- $E'$ : In at most  $D$  steps, we reach a node  $u_0$  that has a long range contact to a node  $u_0^+ \neq n$  such that  $d(u_0, u_0^+) > d$ .
- $E'_i$ : In step  $i$ , we reach a node  $u_0$  that has a long range contact to a node  $u_0^+ \neq n$  such that  $d(u_0, u_0^+) > d$ .

Again, let  $X$  be the random variable which counts the number of steps to reach  $n$  from 0. For every node  $u$ , let  $u^+$  be the long range contact of  $u$ . Using Condition 2 of the lemma, we obtain

$$\Pr(E') = \Pr(\cup_{i=1}^D E'_i) \leq \sum_{i=1}^D \Pr(E'_i) \leq D \cdot \Pr(\{d(u, u^+) > d\}) = D \cdot \sum_{i>d} p(i) \leq \epsilon.$$

Since  $dD < n$ ,  $E \subseteq E'$ , and hence  $\overline{E'} \subseteq \overline{E}$ . Therefore,  $\mathbf{E}[X] \geq \Pr(\overline{E'}) \cdot \mathbf{E}[X | \overline{E'}] \geq (1 - \epsilon)D$ . This completes the proof of the lemma.

**Theorem 4.** *The expected number of steps for greedy routing on  $R_{n+1}$  under the  $r$ -harmonic distribution is bounded from below by (up to a constant):*

$$\begin{cases} n^{\frac{1-r}{2-r}} & \text{if } r < 1 \\ n^{\frac{r-1}{r}} & \text{if } 1 < r \end{cases}$$

*Proof.* The cumulative distributions of the  $r$ -harmonics random variable  $H_r$  are given (up to a multiplicative constant) by the formulas

$$\Pr(\{H_r \leq k\}) \approx \begin{cases} (k/n)^{1-r} & \text{if } r < 1; \\ 1 - k^{1-r} & \text{if } r > 1. \end{cases}$$

When  $r < 1$  we apply condition 1 of Lemma 8 with  $d = D = n^{\frac{1-r}{2-r}}$ , and  $\epsilon = 1/2$ . When  $r > 1$  we apply condition 2 of Lemma 8 with  $d = n^{\frac{1}{r}}$ ,  $D = n^{\frac{r-1}{r}}$ , and  $\epsilon = 1/2$ .

In the specific case  $r = 1$ , one can prove the optimality of Theorem 11 for the 1-harmonic distribution.

**Theorem 5.** *The expected number of steps of greedy routing using the 1-harmonic distribution is at least  $\Omega(\log^2 n)$ .*

*Proof.* Let  $H$  be a 1-harmonic random variable in  $\{1, \dots, n\}$ , i.e.,  $\Pr(\{H = i\}) = 1/(i \cdot H_n)$  where  $H_n = \sum_{i=1}^n 1/i = \Theta(\log n)$ . For any  $s$ ,  $0 \leq s \leq n-1$ , the jump at node  $s$  is  $J_s = \begin{cases} H & \text{if } H \leq n-s; \\ 1 & \text{otherwise.} \end{cases}$  Greedy routing from 0 to  $n$  constructs



a sequence  $s_0 = 0, s_1, s_2, \dots$  such that  $s_{i+1} = s_i + J_{s_i}$ . From Lemma 3, we have, for any  $k \in \{1, \dots, n-s\}$ ,

$$\Pr(\{J_s = k\}) = \begin{cases} \Pr(\{H = k\}) & \text{if } 1 < k \leq n-s; \\ \Pr(\{H = 1\}) + \Pr(\{H > n-s\}) & \text{otherwise.} \end{cases} \quad (2)$$

and

$$\mathbf{E}(J_s) = \Pr(\{H > n-s\}) + \frac{n-s}{H_n} \leq 1 + \frac{n-s}{H_n}. \quad (3)$$

For  $0 \leq i \leq \lfloor \log_2 n \rfloor$ , let  $n_i = n \cdot (1 - 1/2^i)$  and  $I_i = [n_i, n_{i+1})$ . Let  $i > 0$ ,  $s \in I_{i-1}$ , and  $E_s$  be the event that the long range contact of  $s$  is in  $[n_{i+1}, n]$  (i.e., greedy routing from  $s$  to  $n$  “jumps” over  $I_i$ ). We have  $\Pr(E_s) = \Pr(\{J_s \geq n_{i+1} - s\})$ , and thus, thanks to Equation 2,  $\Pr(E_s) = \sum_{k=n_{i+1}-s}^{n-s} \Pr(\{H = k\}) \leq \frac{1}{\log n} \log \left( \frac{n-s}{n_{i+1}-s} \right) = \frac{1}{\log n} \log \left( 1 + \frac{n-n_{i+1}}{n_{i+1}-s} \right)$ . For  $s \in I_{i-1}$ , we have  $2 - \log 3 \leq \log \left( 1 + \frac{n-n_{i+1}}{n_{i+1}-s} \right) \leq 1$ . As a consequence,

$$\Pr(E_s) = \Theta\left(\frac{1}{\log n}\right). \quad (4)$$

Let  $K$  be the random variable defined as the number of consecutive first intervals containing at least one node  $s_i$ , while performing greedy routing from 0 to  $n$ . More precisely, if greedy routing constructs the sequence  $s_0 = 0, s_1, s_2, \dots$ , then  $K = \min\{j : s_i \notin I_j, \forall i\} - 1$ . From Equation 4,

$$\Pr(\{K = k\}) = \Theta\left(\frac{(1 - 1/\log n)^k}{\log n}\right).$$

By using  $\ln(1+x) \sim x$  when  $x$  is small, easy calculations show that  $\mathbf{E}(K) = \Theta(\log n)$ . Let us now concentrate on the time it takes to traverse an interval  $I_i$ ,  $i \leq K$ . Let

$$t_i = \min\{s_j : s_j \in I_i\} \text{ and } t'_i = \max\{s_j : s_j \in I_i\}.$$

Then let  $\Delta_i = t_i - n_i$  and  $\Delta'_i = n_{i+1} - t'_i$ . If  $t_i = s_j$  and  $s_{j-1} \in I_\ell$ , then  $\Delta_i \leq J_{n_\ell} = J_{n(1-1/2^\ell)}$  and thus, thanks to Equation 3,

$$\mathbf{E}(\Delta_i) \leq \mathbf{E}(J_{n(1-1/2^\ell)}) \leq 1 + \frac{n}{2^\ell H_n}.$$

Similarly,

$$\mathbf{E}(\Delta'_i) \leq \mathbf{E}(J_{n(1-1/2^i)}) \leq 1 + \frac{n}{2^i H_n}.$$

Therefore, if  $i \leq K$ , we get  $\ell = i - 1$ , and thus

$$\mathbf{E}(\Delta_i) \leq 1 + \frac{4|I_i|}{H_n} \text{ and } \mathbf{E}(\Delta'_i) \leq 1 + \frac{2|I_i|}{H_n}. \quad (5)$$

Let  $D_i = t'_i - t_i$ . We have  $D_i = (n_{i+1} - n_i) - (\Delta_i + \Delta'_i)$ , and thus from Equation 5

$$\mathbf{E}(D_i) \geq |I_i|(1 - 6/H_n). \quad (6)$$

In the interval  $I_i$ , the long range contacts are at distance at most  $J_{n_i} = J_{n(1-1/2^i)}$ . Let  $X^{(i)} = J_{n(1-1/2^i)}$ , and let  $N_i$  be the stopping time for  $X^{(i)}$ , that is

$$N_i = \min\{k \mid \sum_{j=1}^k X^{(i)} \geq D_i\}.$$

From Equation 6, we have  $\mathbf{E}(\sum_{j=1}^{N_i} X^{(i)}) \geq \mathbf{E}(D_i) \geq |I_i|(1 - 6/H_n)$ . On the other hand, by Wald's Equation (see [15] (Corollary 6.2.3)), we have  $\mathbf{E}(\sum_{j=1}^{N_i} X^{(i)}) = \mathbf{E}(N_i) \cdot \mathbf{E}(X^{(i)})$ . Therefore, from Equation 3 we get

$$\mathbf{E}(N_i) \geq \frac{|I_i|(1 - 6/H_n)}{1 + 2|I_i|/H_n} = \Omega(\log n).$$

To summarize, the expected number of consecutive intervals  $I_i$  traversed by the greedy routing is  $\Omega(\log n)$ , and the expected number of steps to traverse each of these intervals is  $\Omega(\log n)$ . Therefore the expected number of steps of greedy routing is at least  $\Omega(\log^2 n)$ .

It is an open problem whether or not the lower bound of Theorem 5 is valid under any distance invariant distribution on the ring  $R_{n+1}$ . However we note the following general result which is an immediate corollary of Lemma 8.

**Corollary 1.** *Let  $p$  be any non-increasing distance-invariant mapping on  $R_{n+1}$  and  $D < n/4$  an integer such that*

$$\min \left\{ \sum_{i=1}^{O(D)} p(i), \sum_{i=\Omega(n/D)}^n p(i) \right\} \leq O\left(\frac{1}{D}\right).$$

*Then the expected number of steps of greedy routing is in  $\Omega(D)$ .*

## 5 Long Range Contact Graphs

In this section, we show how to generalize the results obtained on the ring to arbitrary graphs. More precisely, we consider the issue of how to produce an appropriate probabilistic mapping  $p$  on an arbitrary graph  $G$  so that routing can be done in a small number of steps in  $(G, p)$ . We begin with the class of  $k$ -dimensional tori.

Kleinberg [11] considers the two dimensional grid. We can generalize his result in the following manner. Consider the  $k$ -dimensional torus  $T_q^k$ : with  $n = q^k$

vertices, i.e.,  $q$  vertices per dimension and  $k \geq 1$ . It is clear that balls of radius  $d$  have size  $\Theta(d^k)$ , and spheres of radius  $d$  have size  $\Theta(d^{k-1})$ . Moreover the diameter is  $D = \Theta(n^{1/k})$ . Let us consider the  $r$ -harmonic distribution on the graph  $T_q^k$ . For the  $r$ -harmonic distribution we have

$$p(d) = \frac{d^{-r}}{\sum_{i=1}^D i^{-r} |S_i(t)|} = \Theta \left( \frac{d^{-r}}{\sum_{i=1}^q i^{-1} \cdot i^{k-r}} \right). \quad (7)$$

Equation (7) indicates that we should select  $r = k$ . In this case we obtain that  $p(d) = \Theta(d^{-k}/\log q)$ . In particular, using Lemma 6,  $(T_q^k, p)$  becomes an  $(f, c)$ -Long Range Contact graph, where

$$\begin{aligned} f(d) &= 1/(p(3d/2) \cdot |B_{d/2}(t)|) \\ &= 1/(\Theta((3d/2)^{-k}/\log q) \cdot (d/2)^{-k}) \\ &= \Theta \left( \frac{3^k}{k} \log n \right). \end{aligned}$$

Since the diameter of the graph  $T_q^k$  is  $D = \Theta(n^{1/k})$  we can use Lemma 4 to obtain the following result.

**Lemma 9.** *Let  $T_q^k$  be the  $k$ -dimensional torus of dimension  $k \geq 1$  and  $n = q^k$  nodes, and let  $p_k$  be the  $k$ -harmonic mapping. Then  $(T_q^k, p_k)$  is an  $(f, 2)$ -Long Range Contact graph, where  $f(d) = \Theta(\frac{3^k}{k} \log n)$ . Moreover, greedy routing in  $(T_q^k, p)$  performs in  $O\left(\frac{3^k}{k^2} \log^2 n\right)$  expected number of steps.*

It follows from Lemma 9 that greedy routing can be performed in  $O(\log^2 n)$  expected number of steps in the  $k$ -dimensional torus  $T_q^k$ , where  $k$  is constant and the probabilistic mapping is defined as before. Let us now present a tool to extend results on a greedy routing in a graph  $G$  to other graphs  $G'$ . First, we recall the notion of an epimorphism.

**Definition 3.** *Consider two graphs  $G = (V, E)$  and  $G' = (V', E')$ . An epimorphism of  $G$  onto  $G'$  is an onto mapping  $\phi : V \rightarrow V'$  such that  $\{u, v\} \in E \Rightarrow \{\phi(u), \phi(v)\} \in E'$ , for all vertices  $u, v \in V$ .*

Note that, if  $\phi$  is an epimorphism, then  $d_{G'}(\phi(u), \phi(v)) \leq d_G(u, v)$  for every  $u$  and  $v$ . Next we define the notion of *distance maintaining* epimorphism.

**Definition 4.** *Let  $\alpha$  be a positive constant. An epimorphism  $\phi$  from the graph  $G = (V, E)$  onto the graph  $G' = (V', E')$  is called  $\alpha$ -distance maintaining if for all  $u, v \in V$ ,  $d_G(u, v) \leq \alpha \cdot d_{G'}(\phi(u), \phi(v))$ . The epimorphism  $\phi$  is called distance maintaining if it is  $\alpha$ -distance maintaining for some positive constant  $\alpha$ .*

It is not hard to see that if  $p$  is a probabilistic mapping on the vertices of  $G$  then  $p'$  is a probabilistic mapping on the vertices of  $G'$ , where

$$p'(u', v') = \frac{1}{|\phi^{-1}(u')|} \sum_{\substack{u \in \phi^{-1}(u') \\ v \in \phi^{-1}(v')}} p(u, v). \quad (8)$$

**Lemma 10.** *Assume that there is an  $\alpha$ -distance maintaining epimorphism  $\phi$  from  $G$  onto  $G'$ . Let  $(G, p)$  be an  $(f, \alpha c)$ -Long Range Contact graph. Then  $(G', p')$  is an  $(f', c)$ -Long Range Contact graph, where  $p'$  is defined in Equation 8 and  $f'(d) = f(\alpha d) \cdot \max_{u' \in V'} |\phi^{-1}(u')|$ .*

*Proof.* Let  $\phi$  be a distance maintaining epimorphism from  $G$  onto  $G'$ . First of all observe that for any  $t, t'$  such that  $\phi(t) = t'$  we have that

$$B_{d'}^{G'}(t') = \{v' : d_{G'}(v', t') \leq d'\} = \{\phi(v) : d_{G'}(\phi(v), \phi(t)) \leq d'\}.$$

Therefore,  $B_{d'}^{G'}(t') \supseteq \phi(\{v : d_G(v, t) \leq d'/c\})$  from the definition of epimorphism. Hence  $B_{d'}^{G'}(t') \supseteq \phi(B_{d'/c}^G(t))$ . It follows that

$$B_{d'}^{G'}(t') \supseteq \bigcup_{t \in \phi^{-1}(t')} \phi(B_{d'/c}^G(t)). \quad (9)$$

Let  $u', t' \in V'$  be vertices such that  $d_{G'}(u', t') \leq d'$ . From the definition of epimorphism, there exist vertices  $u_0, t_0 \in V$  such that  $\phi(u_0) = u', \phi(t_0) = t'$ . Then from the definition of distance maintaining, we have  $d_G(u_0, t_0) \leq \alpha \cdot d_{G'}(u', t') \leq \alpha \cdot d'$ . We have

$$\begin{aligned} p'[u' \rightarrow B_{d'/c}^{G'}(t')] &= \sum_{v' \in B_{d'/c}^{G'}(t')} p'(u', v') \\ &= \frac{1}{|\phi^{-1}(u')|} \sum_{v' \in B_{d'/c}^{G'}(t')} \sum_{u \in \phi^{-1}(u')} \sum_{v \in \phi^{-1}(v')} p(u, v) \\ &\geq \frac{1}{|\phi^{-1}(u')|} \sum_{v' \in B_{d'/c}^{G'}(t')} \sum_{v \in \phi^{-1}(v')} p(u_0, v) \\ &= \frac{1}{|\phi^{-1}(u')|} \sum_{v \in \phi^{-1}(B_{d'/c}^{G'}(t'))} p(u_0, v). \end{aligned}$$

Therefore, from Inequality 9, we get

$$p'[u' \rightarrow B_{d'/c}^{G'}(t')] \geq \frac{1}{|\phi^{-1}(u')|} \sum_{v \in \bigcup_{t \in \phi^{-1}(t')} B_{d'/c}^G(t)} p(u_0, v).$$

It follows that

$$\begin{aligned}
 p'[u' \rightarrow B_{d'/c}^{G'}(t')] &\geq \frac{1}{|\phi^{-1}(u')|} p[u_0 \rightarrow \bigcup_{t \in \phi^{-1}(t')} B_{d'/c}^G(t)] \\
 &\geq \frac{1}{|\phi^{-1}(u')|} p[u_0 \rightarrow B_{d'/c}^G(t_0)] \\
 &\geq \frac{1}{|\phi^{-1}(u')|} p[u_0 \rightarrow B_{(\alpha d')/(\alpha c)}^G(t_0)] \\
 &\geq \frac{1}{|\phi^{-1}(u')|} \frac{1}{f(\alpha d')} \\
 &\geq 1 / \left( f(\alpha d') \cdot \max_{v' \in V'} |\phi^{-1}(v')| \right) \\
 &\geq 1 / f'(d').
 \end{aligned}$$

This completes the proof of the Lemma.

Lemma 10 enables us to define new distributions on graphs.

**Theorem 6.** *Let  $G = (V, E)$  be any graph such that there is a distance maintaining epimorphism  $\phi$  from a  $k$ -dimensional torus of size  $O(n)$  onto  $G$ . Further assume that  $\max_{v \in V} |\phi^{-1}(v)| = O(1)$ . Then there is a probabilistic mapping  $p$  on  $G$  such that greedy routing in  $(G, p)$  performs in  $O(\frac{3^k}{k} \log^2 n)$  expected number of steps.*

*Proof.* From Lemma 9  $(T_q^k, p_k)$  is an  $(f, 2)$ -Long Range Contact graph, where  $f(d) = \Theta(\frac{3^k}{k} \log n)$ . By application of Lemma 8, the probability  $p'$  defined in Equation 8 is such that  $(G, p')$  is an  $(f', 2)$ -Long Range Contact graph where  $f'(d) \leq \beta \cdot f(\alpha d)$  for some constants  $\alpha$  and  $\beta$ . That is  $f'(d) = O(\frac{3^k}{k} \log n)$ . It follows from Lemma 4 that greedy routing in  $(G, p')$  performs in  $O(\frac{3^k}{k} \log n)$  expected number of steps.

## 6 Conclusion and Open Problems

In this paper we have studied the performance of greedy routing in the ring augmented with long range contacts chosen using  $r$ -harmonic distributions. We have also shown how to extend our results to arbitrary networks via appropriate mappings of multidimensional tori onto the network. Under certain conditions it is shown that greedy routing performs quite efficiently, i.e.,  $O(\log^2 n)$  expected number of steps. In particular, the ring augmented with the 1-harmonic distribution provides a simple model for the small world phenomenon.

Several interesting problems remain. For a general network, can we define probabilistic mappings for which greedy routing has better performance? Is our

$\Omega(\log^2 n)$  lower bound on the ring valid for all distance invariant mappings (not just the  $r$ -harmonic) on the  $n$ -node ring? Similar questions apply to any multidimensional torus. We note that in this paper we emphasized greedy routing, in the sense that nodes forward messages to their neighbors which are closer to the destination. An interesting open problem is to study the resulting tradeoff between memory (required at the nodes of the network) and type of routing being used.

## References

1. W. Aiello, F. Chung, and L. Lu, "A Random Graph Model for Massive Graphs", in 32nd Annual ACM Symposium on Theory of Computing (STOC), 2000.
2. B. Bollobas, "Random graphs", Academic Press, 1985.
3. B. Bollobas and F. Chung, "The Diameter of a Cycle Plus a Random Matching", SIAM Journal on Discrete Mathematics 1:328-333, 1988.
4. A. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, J. Wiener, "Graph Structure in the Web", in 9th International World Wide Web Conference, Amsterdam, May 15-19, 2000.
5. F. Chung and M. Garey, "Diameter Bounds for Altered Graphs", Journal of Graph Theory, 8:511-534, 1984.
6. M. Faloutsos, P. Faloutsos, C. Faloutsos, "On Power-Law Relationships of the Internet Topology", in ACM annual Technical Conference of the Special Interest Group on Data Communication (SIGCOMM), 1999.
7. B. Hayes, "Graph Theory in Practice: Part I", American Scientist, January-February, Vol. 88, No. 1, 2000.
8. B. Hayes, "Graph Theory in Practice: Part II", American Scientist, March-April, Vol. 88, No. 2, 2000.
9. R. Karp, "Probabilistic Recurrence Relations", in Proceedings of 23rd Annual Symposium on Theory of Computing, pages 190-197, 1991.
10. R. Karp, E. Upfal, and A. Wigderson, "Constructing a Perfect Matching is in Random NC", Combinatorica, 6:35-48, 1986.
11. J. Kleinberg, "The Small-World Phenomenon: An Algorithmic Perspective", in 32nd Annual ACM Symposium on Theory of Computing (STOC), 2000.
12. J. Kleinberg, R. Kumar, P. Raghavan, S. Rajagopalan, and A. Tomkins, "The Web as a Graph: Measurements, Models, and Methods", in 5th International Conference on Computing and Combinatorics, Tokyo, July 26-28, 1999.
13. D. Knuth, "The Art of Computer Programming: Sorting and Searching", Vol. 3, 2nd edition, Addison-Wesley, 1998.
14. R. Motwani and P. Raghavan, "Randomized Algorithms", Cambridge University Press, 1995.
15. S. Ross, "Stochastic Processes", 2nd ed., John Wiley and Sons, 1996.
16. D. Watts, "Small World: The Dynamics of Networks between Order and Randomness", Princeton University Press, 1999.
17. D. Watts and S. Strogatz, "Collective Dynamics of Small-World Networks", Nature 393:440-442, 1998.
18. G. K. Zipf, "Human Behavior and the Principle of Least Effort", Addison Wesley, Cambridge MA, 1949, Reprint New York, Hefner, 1972.

# An Efficient Communication Strategy for Ad-hoc Mobile Networks<sup>\*</sup>

I. Chatzigiannakis, S. Nikolettseas, and P. Spirakis

<sup>1</sup> 1. Computer Technology Institute, Patras, Greece  
{ichatz,nikole,spirakis}@cti.gr

<sup>2</sup> 2. Computer Engineering and Informatics Department,  
Patras University, Greece

**Abstract.** We investigate the problem of communication in an *ad-hoc* mobile network, that is, we assume the extreme case of a total absence of any fixed network infrastructure (for example a case of rapid deployment of a set of mobile hosts in an unknown terrain). We propose, in such a case, that a *small* subset of the deployed hosts (which we call the *support*) should be used for network operations. However, the vast majority of the hosts are moving *arbitrarily* according to application needs.

We then provide a simple, correct and efficient protocol for communication that avoids message flooding. Our protocol manages to establish communication between *any pair* of mobile hosts in *small, a-priori guaranteed expected time bounds* even in the *worst* case of *arbitrary* motions of the hosts that not in the support (provided that they do not deliberately try to avoid the support). These time bounds, interestingly, *do not depend*, on the number of mobile hosts that do not belong in the support. They depend only on the size of the area of motions. Our protocol can be implemented in very efficient ways by exploiting knowledge of the space of motions or by adding more power to the hosts of the support.

Our results exploit and further develop some fundamental properties of random walks in finite graphs.

## 1 Introduction

**Ad-hoc Mobile Networks:** An ad-hoc mobile network ([12,1]) is a collection of mobile hosts with wireless network interfaces forming a temporary network *without the aid of any established infrastructure or centralised administration*. In an ad-hoc network two hosts that want to communicate may not be within wireless transmission range of each other, but could communicate if other hosts between them in the ad-hoc network are willing to forward packets for them.

A *basic communication problem*, in such networks, is to send information from some *sender* user, *S*, to another designated *receiver* user, *R*. Remark that ad-hoc mobile networks are dynamic in nature, in the sense that local connections are temporary and may change as users move. The movement rate of each user might

---

<sup>\*</sup> This work was partially supported by the EU projects IST FET-OPEN ALCOM-FT, IMPROVING RTN ARACNE and the Greek GSRT Project PENED99-ALKAD.

vary, while certain hosts might even stop (even in “remote” areas) in order to execute location-oriented tasks (e.g. take measurements).

A protocol solving this important communication problem is *reliable* if it allows the sender to be notified about delivery of the information to the receiver.

**The innovation and justification of our approach:** One way to solve this problem is the protocol of notifying every user that the sender meets (and providing *all the information to it*) hoping that some of them will eventually meet the receiver.

Is there a more efficient technique that will effectively solve the communication problem without flooding the network and exhausting the battery and computational power of the hosts?

The most common way to establish communication is to form paths of intermediate nodes that lie within one another’s transmission range and can directly communicate with each other [13,15,16]. Indeed, this approach of exploiting pairwise communication is common in ad-hoc mobile networks that cover a relatively small space (i.e. with diameter which is small with respect to transmission range) or are dense (i.e. thousands of wireless nodes) where all locations are occupied by some hosts; broadcasting can be efficiently accomplished.

In wider area ad-hoc networks with less users, however, broadcasting is impractical: two distant peers will not be reached by any broadcast as users may not occupy all intermediate locations (i.e. the formation of a path is not feasible). Even if a valid path is established, single link “failures” happening when a small number of users that were part of the communication path move in a way such that they are no longer within transmission range of each other, will make this path invalid. Note also that the path established in this way may be very long, even in the case of connecting nearby hosts.

In contrast to all such methods, we try to avoid ideas based on paths finding and their maintenance. We envision networks with highly dynamic movement of the mobile users, where the idea of “maintenance” of a valid path is inconceivable (paths can become invalid immediately after they have been added to the directory tables). Our approach is to take advantage of the mobile hosts natural movement by exchanging information whenever mobile hosts meet incidentally. It is evident, however, that if the users are spread in remote areas and they do not move beyond these areas, there is no way for information to reach them, unless the protocol takes special care of such situations.

In the light of the above, we propose the idea of forcing only a small subset of the deployed hosts to move as per the needs of the protocol. Assuming the availability of such hosts, we use them to provide a simple, correct and efficient strategy for communication between any pair of hosts in such networks that avoid message flooding.

**A scenario for rapid deployment of mobile hosts:** A usual scenario that fits to the ad-hoc mobile model is the particular case of rapid deployment of mobile hosts, in an area where there is no underlying fixed infrastructure (either because it is impossible or very expensive to create such an infrastructure, or



because it is not established yet, or it has become temporarily unavailable i.e. destroyed or down).

In such a case of rapid deployment of a number of mobile hosts, it is possible to have a small team of fast moving and versatile vehicles, to implement the support. These vehicles can be cars, jeeps, motorcycles or helicopters. We interestingly note that this small team of fast moving vehicles can also be a collection of independently controlled mobile modules, i.e. robots. This specific approach is inspired by the recent paper of J.Walter, J.Welch and N.Amato. In their paper “Distributed Reconfiguration of Metamorphic Robot Chains” ([17]) the authors study the problem of motion co-ordination in distributed systems consisting of such robots, which can connect, disconnect and move around. The paper deals with metamorphic systems where (as is also the case in our approach) all modules are identical. Note that the approach of having the support moving in a co-ordinated way, *i.e. as a chain of nodes*, has some similarities to [17].

**Our results:** We provide a particular protocol (and a specific support coordination subprotocol) which guarantees correct and efficient communication for any pair of users, in (expected) time *depending only on the size of the network area, independently of the motion of the hosts not in the support and independently of their number*. We achieve this by assuming that a *small* part of the deployed hosts, which we call the support, can move fast in a *coordinated* way, to *sweep* the motion space and act as an *intermediate pool* for receiving and delivering messages to the mobile users.

In a way similar to [17], these moving modules are identical in computing and communication (i.e. transmission) capability and run the same support management subprotocol to determine movement and communication of the hosts in the support. Furthermore, note that each module in the support needs only to know its current location (i.e. only local information is needed and not a global picture of the entire area). However, additional global information (such as knowledge of a spanning subgraph of the motion space) can improve the performance of our protocol.

Our protocol is simple, scalable, does not assume common sense of orientation, and does not need a lot of memory. It is resilient to single-host failures of the support. Furthermore, our protocol *avoids* the problem of flooding the network with messages.

The proof of our main theorem exploits the fundamental notion of strong stationary times of reversible Markov Chains. This notion allows us to consider general motion strategies of the users not in the support.

In [4] we performed extensive experiments (and some analysis) of a version of such a strategy but without the general framework and only for the *restricted case where all users (even those not in the support) perform independent and concurrent random walks*. A model for motion (without geometry details) for mobile networks was introduced by members of our team in [11]. Related material has appeared as a brief announcement in the Proceedings of the 20th Annual Symposium on Principles of Distributed Computing, [6]. For a survey of selected

work in distributed communication and control issues in ad-hoc mobile networks, see [7].

**Previous Work:** In a recent paper [14], Q.Li and D.Rus present a model which has some similarities to ours. The authors give an interesting, yet different, protocol to send messages, which forces *all the mobile hosts to slightly deviate (for a short period of time) from their predefined, deterministic routes, in order to propagate the messages*. Their protocol is, thus, *compulsory* for any host and it works only for deterministic host routes. Moreover, their protocol considers the propagation of only one message (end to end) each time, in order to be correct. In contrast, our support scheme allows for simultaneous processing of many communication pairs. In their setting [14] show optimality of message transmission times.

M.Adler and C.Scheideler [1] in a previous work, dealt only with *static* transmission graphs i.e. the situation where the positions of the mobile hosts and the environment do not change. In [1] the authors pointed out that static graphs provide a starting point for the dynamic case. In our work, we consider the *dynamic case* (i.e. mobile hosts move *arbitrarily*) and in this sense we extend their work. As far as performance is concerned, their work provides time bounds for communication that are proportional to the diameter of the graph defined by random uniform spreading of the hosts, while our time bounds are linear to the area of motions, and *independent of the number of mobile hosts, or their spreading*.

We quantify our protocol's performance (in terms of communication *time*) and we show how to make it efficient and how to estimate the best size of the support.

## 2 The Model of the Space of Motions

Based on the work of [11, 4] we abstract the environment where the stations move (in three-dimensional space with possible obstacles) by a *motion-graph* (i.e. we neglect the detailed geometric characteristics of the motion). In particular, we first assume that each mobile host has a transmission range represented by a sphere  $tr$  centred by itself. We approximate this sphere by a cube  $tc$  with volume  $\mathcal{V}(tc)$  the maximum such that  $\mathcal{V}(tc) < \mathcal{V}(tr)$ . Given that the mobile hosts are moving in the space  $\mathcal{S}$ ,  $\mathcal{S}$  is divided into consecutive cubes of volume  $\mathcal{V}(tc)$ .

**Definition 1.** *The motion graph  $G(V, E)$ , ( $|V| = n$ ,  $|E| = m$ ), which corresponds to a quantization of  $\mathcal{S}$  is constructed in the following way: a vertex  $u \in G$  represents a cube of volume  $\mathcal{V}(tc)$ . An edge  $(u, v) \in G$  if the corresponding cubes are adjacent.*

The number of vertices  $n$ , actually approximates the ratio between the volume of space  $\mathcal{S}$ ,  $\mathcal{V}(\mathcal{S})$ , and the space occupied by the transmission range of a mobile host  $\mathcal{V}(tr)$ . Given the transmission range  $tr$ ,  $n$  depends linearly on the volume of space  $\mathcal{S}$  regardless of the choice of  $tc$ , and  $n = O\left(\frac{\mathcal{V}(\mathcal{S})}{\mathcal{V}(tr)}\right)$ . Let us call

the ratio  $\frac{V(\mathcal{S})}{V(tr)}$  by the term *relative motion space size* and denote it by  $\rho$ . Since the edges of  $G$  represent neighbouring polyhedra each node is connected with a constant number of neighbours, which yields that  $m = \Theta(n)$ . Let  $\Delta$  be the maximum vertex degree of  $G$ .

### 3 A Protocol Framework for Ad-hoc Mobile Networks

We wish to look into ad-hoc networks where a small part of their hosts is used to serve network needs for communication. This is captured by the following:

**Definition 2.** *The class of ad-hoc mobile network protocols which enforce a (small) subset of the mobile hosts to move in a certain way is called the class of semi-compulsory protocols.*

**Definition 3.** *The subset of the mobile hosts of an ad-hoc mobile network whose motion is determined by a network protocol  $\mathcal{P}$  is called the support  $\Sigma$  of  $\mathcal{P}$ . The part of  $\mathcal{P}$  which indicates the way that members of  $\Sigma$  move and communicate is called the support management subprotocol of  $\mathcal{P}$ .*

**Definition 4.** *Consider a family of protocols,  $\mathcal{F}$ , for a mobile ad-hoc network, and let each  $\mathcal{P}$  in  $\mathcal{F}$  have the same support (and the same support management subprotocol). Then  $\Sigma$  is called the support of the family  $\mathcal{F}$ .*

In addition, we may wish that the way hosts in  $\Sigma$  move (maybe coordinated) and communicate is robust (i.e. can tolerate failures of hosts).

The types of failures of hosts that we consider here are permanent (i.e. stop) failures.

**Definition 5.** *A support management subprotocol,  $M_\Sigma$ , is  $k$ -faults tolerant, if it still allows the members of  $\mathcal{F}$  (or  $\mathcal{P}$ ) to execute correctly, under the presence of at most  $k$  permanent faults of hosts in  $\Sigma$  ( $k \geq 1$ ).*

We assume, that the motions of the mobile users which are not members of  $\Sigma$  are arbitrary but *independent* of the motion of the support (i.e. we exclude the case where some of the users not in  $\Sigma$  are deliberately trying to avoid  $\Sigma$ ). This is a pragmatic assumption usually followed by application protocols. We call it the *independence assumption*.

**Definition 6.** *A ad-hoc mobile network is not hostile if the hosts not in  $\Sigma$  obey the independence assumption.*

## 4 Our Proposed Strategy

### 4.1 The Scheme

Our proposed scheme, in simple terms, works as follows: The nodes of the support move fast enough in a coordinated way so that they sweep (in sufficiently short time) the entire motion graph. Their motion and communication is accomplished in a distributed way via a *support management subprotocol*  $M_\Sigma$ . When some node of the support is within communication range of a sender, an underlying *sensor subprotocol*  $M_\Sigma^l$  notifies the sender that it may send its message(s).

The messages are then stored “somewhere within the support structure”. For simplicity we may assume that they are copied and stored in every node of the support. This is not the most efficient storage scheme and can be refined in various ways. When a receiver comes within communication range of a node of the support, the receiver is notified that a message is “waiting” for him and the message is then forwarded to the receiver. For simplicity, we will also assume that message exchange between nodes within communication distance of each other takes negligible time. Note that this general scheme allows for easy implementation of many-to-one communication and also multicasting. In a way, the support  $\Sigma$  plays the role of a (moving) skeleton subnetwork (of a “fixed” structure, guaranteed by the motion subprotocol  $M_\Sigma$ ), through which all communication is routed. From the above description, the size,  $k$ , and the shape of the support may affect performance.

Our scheme follows the general design principle of mobile networks (with a fixed subnetwork however) called the “two-tier” principle ([12]) which says that any protocol should try to move communication and computation to the fixed part of the network. Our idea of the support  $\Sigma$  is a simulation of such a (skeleton) network by moving hosts, however.

Note that the proposed scheme does not require the propagation of messages through hosts that are not part of  $\Sigma$ , thus its security relies on the support’s security and is not compromised by the participation in message communication of other mobile users. For a discussion of intrusion detection mechanisms for ad-hoc mobile networks see [18].

### 4.2 The Implementation Proposed for $\Sigma$ , $M_\Sigma$

There is a set-up phase of the ad-hoc network, where a predefined set,  $k$ , of hosts, become the nodes of the support. The members of the mobile support perform a leader election by running a randomized symmetry breaking protocol in anonymous networks ([11]). This imposes only an initial communication cost. The elected leader, denoted by  $MS_0$ , is used to co-ordinate the support topology and movement. Additionally, the leader assigns local names to the rest of the support members ( $MS_1, MS_2, \dots, MS_{k-1}$ ). The movement of  $\Sigma$  is then defined as follows:

Initially,  $MS_i, \forall i \in \{0, 1, \dots, k-1\}$ , start from the same area-node of the motion graph. The direction of movement of the leader  $MS_0$  is

given by a memoryless operation that chooses *randomly the direction* of the next move. Before leaving the current area-node,  $MS_0$  sends a message to  $MS_1$  that states the new direction of movement.  $MS_1$  will change its direction as per instructions of  $MS_0$  and will propagate the message to  $MS_2$ . In analogy,  $MS_i$  will follow the orders of  $MS_{i-1}$  after transmitting the new directions to  $MS_{i+1}$ . Movement orders received by  $MS_i$  are positioned in a queue  $Q_i$  for sequential processing. The very first move of  $MS_i$ ,  $\forall i \in \{1, 2, \dots, k-1\}$  is delayed by  $\delta$  period of time.

We assume that the mobile support hosts move with a common speed. Note that the above described motion subprotocol  $M''_\Sigma$  enforces the support to move as a “snake”, with the head (the elected leader  $MS_0$ ) *doing a random walk on the motion graph  $G$*  and each of the other nodes  $MS_i$  executing the simple protocol “move where  $MS_{i-1}$  was before”. Therefore our protocol does not require common sense of orientation.

The purpose of the random walk of the head is to ensure a *cover* (within some finite time) of the whole motion graph, without memory (other than local) of topology details. Note that this memoryless motion also ensures fairness.

A modification of  $M_\Sigma$  is that the head does a random walk on a *spanning subgraph* of  $G$  (eg. a spanning tree). This modified  $M_\Sigma$  (call it  $T_\Sigma$ ) is more efficient in our setting since “edges” of  $G$  just represent adjacent locations and “nodes” are really possible host places.

### 4.3 Alternative Implementations - Extensions

One can think also of other ways to implement the support management subprotocol  $M_\Sigma$ :

- The *runners* implementation of  $M_\Sigma$  allows each member of  $\Sigma$  to move via an *independent* random walk (on the same spanning subgraph of  $G$ ). When runners meet, they exchange information given to them by hosts. This management subprotocol provides improved reliability in the sense that it is resilient to  $t$  faults, where  $t < k$ . However, note that messages may have to be re-transmitted in the case that only one copy of them exists when the faults occur.

The key observation justifying this approach (and maybe its superiority, with respect to performance, compared to the “snake” approach) is that each runner will meet each other in parallel, thus accelerating the spread of information. In [8] we experimentally showed that the “runners” protocol outperforms the “snake” protocol.

- In *hierarchical* motion graphs [5] we can divide  $\Sigma$  into a subset  $\Sigma'$  moving only in the upper level of the hierarchy and the hosts of  $\Sigma - \Sigma'$  which can be split in “snakes”, each randomly walking inside the lower levels of the hierarchy. The lower level of the hierarchy may model dense ad-hoc subnetworks of mobile users that are unstructured and where there is no fixed infrastructure. To implement communication in such a case, a possible solution would be to install a very fast (yet limited) backbone interconnecting such highly populated mobile user areas,

while using the support approach in the lower levels. This fast backbone provides a limited number of access ports within these dense areas of mobile users.

In such hierarchical cases communication between users in different dense areas takes place in the following way: The support first gets from the sender node the messages upon meeting him and conveys these messages to the backbone system when meeting the corresponding access port. Then by exploiting the very fast communication over the backbone,  $\Sigma_1$  forwards the messages to some access port in the receiver area, from which subsequently the messages are picked by the local support ( $\Sigma_2$ ) and delivered to the receiver host.

We note that this hierarchical approach for a management subprotocol is inherently *modular*.

#### 4.4 Protocol Correctness Properties

In the sequel we investigate non-hostile ad-hoc mobile networks. We assume that each mobile host has sufficient power supplies (or on-line power feedings) to support communication for long times. Moreover, we assume (to simplify the technical analysis) common speed and fixed transmission range for the hosts not in the support.

In the sequel, we assume that the head of  $\Sigma$  does a *continuous time random walk* on  $G(V, E)$ , without loss of generality (we can discretize). We define the random walk of a mobile user on  $G$  that induces a continuous time Markov chain  $M_G$  as follows: The states of  $M_G$  are the vertices of  $G$ . Let  $s_t$  denote the state of  $M_G$  at time  $t$ . Given that  $s_t = u$ ,  $u \in V$ , the probability that  $s_{t+dt} = v$ ,  $v \in V$ , is  $p(u, v) \cdot dt$  where

$$p(u, v) = \begin{cases} \frac{1}{d(u)} & \text{if } (u, v) \in E \\ 0 & \text{otherwise} \end{cases}$$

and  $d(u)$  is the degree of vertex  $u$ .

**Definition 7.**  $P_i(E)$  is the probability that the walk satisfies an event  $E$  given it started at vertex  $i$ .

**Definition 8.** For a vertex  $j$ , let  $T_j$  be the first hitting time of the walk onto that vertex and let  $E_i T_j$  be its expected value, given that the walk started at vertex  $i$  of  $G$ .

**Definition 9.** For the walk of  $\Sigma$ 's head, let  $\pi()$  be the stationary distribution of its position after a sufficiently long time.

We know (see [2]) that for every vertex  $\sigma$ ,  $\pi(\sigma) = \frac{d(\sigma)}{2m}$  where  $d(\sigma)$  is the degree of  $\sigma$  in  $G$  and  $m = |E|$ .

**Definition 10.** Let  $p_{j,k}$  be the transition probability of the walk of  $\Sigma$ 's head from vertex  $j$  to vertex  $k$ . Let  $p_{j,k}(t)$  be the probability that the walk started at  $j$  will be at  $k \in V$  in time  $t$ .

**Theorem 1.** *The support  $\Sigma$  and the management subprotocol  $M_\Sigma$  guarantee reliable communication establishment between any sender-receiver  $(S, R)$  pair in finite time, whose expected value is bounded only by a function of the relative motion space size  $\rho$  and does not depend on the number of hosts, and is also independent of how  $S, R$  move.*

*Proof.* Any sender  $S$  or receiver  $R$  is allowed an *arbitrary* strategy of motion but it does not deliberately try to avoid the support  $\Sigma$ . So, it either executes a deterministic motion (which either stops at a node, or repeats forever) or follows a random strategy *independent* of the random walk of the support's head.

For the proof purposes, it is enough to show that the head of  $\Sigma$  will meet  $S$  and  $R$  infinitely often, with probability 1 (in fact our argument is a consequence of the Borel-Cantelli Lemmas for infinite sequences of trials). We will furthermore show that the first meeting time  $M$  (with  $S$  or  $R$ ) has an expected value (where expectation is taken over the walk of  $\Sigma$  and any strategy of  $S$  (or  $R$ ) and any starting position of  $S$  (or  $R$ ) and  $\Sigma$ ) which is bounded by a function of the size of the motion graph  $G$  only. This then shows the Theorem since it shows that  $S$  (and  $R$ ) meet with the head of  $\Sigma$  infinitely often, each time within a bounded expected duration.

So, let  $EM$  be the expected time of the (first) meeting and  $m^* = \sup EM$ , where the supremum is taken over all starting positions of both  $\Sigma$  and  $S$  (or  $R$ ) and all strategies of  $S$  (one can repeat the argument with  $R$ ).

We will now assume w.l.o.g. (see [2]) that the head of  $\Sigma$ 's walk is a continuous-time random walk on  $G$ . The states of the walk of  $\Sigma$ 's head are just the vertices of  $G$  and they are finite.

**Definition 11.** *Let  $X(t)$  be the position of the walk at time  $t$*

We proceed to show that we can construct for the walk of  $\Sigma$ 's head a *strong stationary time sequence*  $V_i$  such that for all  $\sigma \in V$  and for all times  $t$

$$P_i(X(V_i) = \sigma \mid V_i = t) = \pi(\sigma)$$

Notice that at times  $V_i$ ,  $S$  (or  $R$ ) will necessarily be at some vertex  $\sigma$  of  $V$ , either still moving or stopped. Let  $u$  be a time such that for  $X$ ,

$$p_{j,k}(u) \geq \left(1 - \frac{1}{e}\right) \pi(k)$$

for all  $j, k$ . Such a  $u$  always exists because  $p_{j,k}(t)$  converges to  $\pi(k)$  from basic Markov Chain Theory. Note that  $u$  depends only on the structure of the walk's graph,  $G$ . In fact, if one defines separation from stationarity to be

$$s(t) = \max_j s_j(t)$$

where

$$s_j(t) = \sup \{s : p_{ij}(t) \geq (1 - s)\pi_j\}$$

then

$$\tau_1^{(1)} = \min\{t : s(t) \leq e^{-1}\}$$

is called the *separation threshold time*. For general graphs  $G$  of  $n$  vertices this quantity is known to be  $\mathcal{O}(n^3)$  ([3]).

Now consider a sequence of stopping times  $U_i \in \{u, 2u, 3u, \dots\}$  such that

$$P_i(X(U_i) = \sigma \mid U_i = u) = \left(1 - \frac{1}{e}\right)\pi(\sigma) \quad (1)$$

for any  $\sigma \in V$ . By induction on  $\lambda \geq 1$  then

$$P_i(X(U_i) = \sigma \mid U_i = \lambda u) = e^{-(\lambda-1)} \left(1 - \frac{1}{e}\right)\pi(\sigma)$$

This is because of the following: First remark that for  $\lambda = 1$  we get the definition of  $U_i$ . Assume that the relation holds for  $(\lambda - 1)$  i.e.

$$P_i(X(U_i) = \sigma \mid U_i = (\lambda - 1)u) = e^{-(\lambda-2)} \left(1 - \frac{1}{e}\right)\pi(\sigma)$$

for any  $\sigma \in V$ . Then  $\forall \sigma \in V$

$$\begin{aligned} P_i(X(U_i) = \sigma \mid U_i = \lambda u) &= \sum_{\alpha \in V} P_i(X(U_i) = \alpha \mid U_i = (\lambda - 1)u) \cdot P_{\alpha, \sigma}(u) \\ &= e^{-(\lambda-2)} \left(1 - \frac{1}{e}\right) \sum_{\alpha \in V} \pi(\alpha) \frac{1}{e} \pi(\sigma) \quad \text{from (I)} \\ &= e^{-(\lambda-1)} \left(1 - \frac{1}{e}\right) \pi(\sigma) \end{aligned}$$

which ends the induction step. Then, for all  $\sigma$

$$P_i(X(U_i) = \sigma) = \pi(\sigma) \quad (2)$$

and

$$E_i U_i = u \left(1 - \frac{1}{e}\right)^{-1}$$

Now let  $c = u \frac{e}{e-1}$ . So, we have constructed (by (2)) a *strong stationary time sequence*  $U_i$  with  $EU_i = c$ . Consider the sequence  $0 = U_0 < U_1 < U_2 < \dots$  such that for  $i \geq 0$

$$E(U_{i+1} - U_i \mid U_j, j \leq i) \leq c$$



But, from our construction, the positions  $X(U_i)$  are *independent* (of the distribution  $\pi()$ ), and, in particular,  $X(U_i)$  are *independent* of  $U_i$ . Therefore, regardless of the strategy of  $S$  (or  $R$ ) and because of the independence assumption, the support's head has chance at least  $\min_{\sigma} \pi(\sigma)$  to meet  $S$  (or  $R$ ) at time  $U_i$ , independently as  $i$  varies. So, the meeting time  $M$  satisfies  $M \leq U_T$  where  $T$  is a stopping time with mean

$$ET \leq \frac{1}{\min_{\sigma} \pi(\sigma)}$$

Note that the idea of a stopping time  $T$  such that  $X(T)$  has distribution  $\pi$  and is independent of the starting position is central to the standard modern Theory of Harris - recurrent Markov Chains (see e.g. [10]).

From Wald's inequality ([2]) then  $EU_T \leq c \cdot ET$ , thus

$$m^* \leq c \frac{1}{\min_{\sigma} \pi(\sigma)}$$

Note that since  $G$  is produced as a subgraph of a regular graph of fixed degree  $\Delta$  we have

$$\frac{1}{2m} \leq \pi(\sigma) \leq \frac{1}{n}$$

for all  $\sigma$  ( $n=|V|$ ,  $m=|E|$ ), thus  $ET \leq 2m$ , hence

$$m^* \leq 2mc = \frac{e}{e-1} 2mu$$

Since  $m, u$  only depend on  $G$ , this proves the Theorem. □

**Corollary 1.** *If  $\Sigma$ 's head walks randomly in a regular spanning subgraph of  $G$ , then  $m^* \leq 2cn$ .*

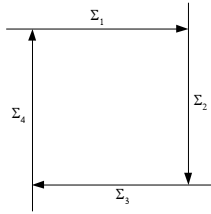
Now, we examine the robustness of the motion management subprotocol under single stop-faults.

**Theorem 2.** *The support management subprotocol  $M_{\Sigma}$  is 1-fault tolerant.*

*Proof.* If a single host of  $\Sigma$  fails, then the following host becomes the head of the rest of the “snake”. We thus have two, independent, random walks in  $G$  (of the two “snakes”) which, however, will meet in expected time at most  $m^*$  (as in Theorem 1) and re-organize as a single snake via a very simple re-organization protocol which is the following:

When the head of the second snake  $\Sigma_2$  meets a host  $h$  of the first snake  $\Sigma_1$  then the head of  $\Sigma_2$  follows the host which is “in front” of  $h$  in  $\Sigma_1$ , and all the part of  $\Sigma_1$  after and including  $h$  waits, to follow  $\Sigma_2$ 's tail. □

Note that in the case when more than one faults occur, the procedure for merging “snakes” described above may lead to deadlock, as figure 4 graphically depicts.



**Fig. 1.** Deadlock situation arising when four “snakes” are about to merge.

#### 4.5 Protocol Time Efficiency Properties

**Crude bounds.** Clearly, one intuitively expects that if  $k = |\Sigma|$  then the higher  $k$  is (with respect to  $n$ ), the best the performance of  $\Sigma$  gets.

By working as in the proof of Theorem 1, we can create a sequence of strong stationary times  $U_i$  such that  $X(U_i) \in F$  where  $F = \{\sigma : \sigma \text{ is a position of a host in the support}\}$ . Then  $\pi(\sigma)$  is replaced by  $\pi(F)$  which is just  $\pi(F) = \sum \pi(\sigma)$  over all  $\sigma \in F$ . So now  $m^*$  is bounded as follows:

$$m^* \leq c \frac{1}{\min_{\sigma \in J} \left( \sum \pi(\sigma) \right)}$$

where  $J$  is any induced subgraph of the graph of the walk of  $\Sigma$ 's head such that  $J$  is the neighbourhood of a vertex  $\sigma$  of radius (maximum distance simple path) at most  $k$ . The quantity

$$\min_J \left( \sum_{\sigma \in J} \pi(\sigma) \right)$$

is then at least  $\frac{k}{2m}$  and, hence,  $m^* \leq c \frac{2m}{k}$ .

Since the communication establishment time,  $T_c$ , between  $S$ ,  $R$  is bounded above by  $X+Y+Z$ , where  $X$  is the time for  $S$  to meet  $\Sigma$ ,  $Y$  is the time for  $R$  to meet  $\Sigma$  (after  $X$ ) and  $Z$  is the message propagation time in  $\Sigma$ , we have for all  $S$ ,  $R$

$$E(T_c) \leq \frac{2mc}{k} + \Theta(k) + \frac{2mc}{k}$$

(since  $Z = \Theta(k)$ ). The upper bound achieves a minimum when  $k = \sqrt{2mc}$ .

**Lemma 1.** *For the walk of  $\Sigma$ 's head on the entire motion graph  $G$ , the communication establishment time's expected time is bounded above by  $\Theta(\sqrt{mc})$  when the (optimal) support size  $|\Sigma|$  is  $\sqrt{2mc}$  and  $c$  is  $\frac{e}{e-1}u$ ,  $u$  being the “separation threshold time” of the random walk on  $G$ .*

**Tighter bounds - improved protocol.** To make our protocol more efficient, we now force the head of  $\Sigma$  to perform a random walk on a *regular spanning graph* of  $G$ . Let  $G_R(V, E')$  be such a subgraph. Our improved protocol versions assume that (a) such a subgraph exists in  $G$  and (b) is given in the beginning to all the stations of the support. By studying, in a way similar to Theorem 1 and [2], the first meeting times and the separation from stationarity of the random walk on the regular spanning graph, we get the following theorem (for the proof see [97]) :

**Theorem 3.** *By having  $\Sigma$ 's head to move on a regular spanning subgraph of  $G$ , there is an absolute constant  $\gamma > 0$  such that the expected meeting time of  $S$  (or  $R$ ) and  $\Sigma$  is bounded above by  $\gamma \frac{n^2}{k}$ .*

Remark again that the total expected communication establishment time is bounded above by  $2\gamma \frac{n^2}{k} + \Theta(k)$  and by choosing  $k = \sqrt{2\gamma n^2}$  we can get a best bound of  $\Theta(n)$  for a support size of  $\Theta(n)$ .

**Corollary 2.** *By forcing the support's head to move on a regular spanning subgraph of the motion graph, our protocol guarantees a total expected communication time of  $\Theta(\rho)$ , where  $\rho$  is the relative motion space size, and this time is independent of the total number of mobile hosts, and their movement.*

Note also that our analysis assumed that the head of  $\Sigma$  moves according to a continuous time random walk of total rate 1 (rate of exit out of a node of  $G$ ). If we select the support's hosts to be  $\psi$  times faster than the rest of the hosts, all the estimated times, except of the inter-support time, will be divided by  $\psi$ . Thus

**Corollary 3.** *Our modified protocol where the support is  $\psi$  times faster than the rest of the mobile hosts guarantees an expected total communication time which can be made to be as small as  $\Theta(\gamma \frac{\rho}{\sqrt{\psi}})$  where  $\gamma$  is an absolute constant.*

## 5 A Lower Bound

**Lemma 2.**

$$m^* \geq \max_{i,j} E_i T_j$$

*Proof.* Consider the case where  $S$  (or  $R$ ) just stands still on some vertex  $j$  and  $\Sigma$ 's head starts at  $i$ . □

**Corollary 4.** *When  $\Sigma$  starts at positions according to the stationary distribution  $\pi$  of its head's walk then,  $\forall j$ ,*

$$m^* \geq \max_j E_\pi T_j$$

□

From a Lemma of ([2], ch. 4, pp. 21), we know that for all  $i$

$$E_{\pi} T_i \geq \frac{(1 - \pi_i)^2}{q_i \pi_i}$$

where  $q_i = d_i$  is the degree of  $i$  in  $G$  i.e.,

$$E_{\pi} T_i \geq \min_i \frac{(1 - \frac{d_i}{2m})^2}{\frac{d_i}{2m}} \geq \min_i \frac{1}{2m} \frac{(2m - d_i)^2}{d_i^2}$$

For regular spanning subgraphs of  $G$  of degree  $\Delta$  we have  $m = \frac{\Delta n}{2}$ , where  $d_i = \Delta$  for all  $i$ . Thus,

**Theorem 4.** *When  $\Sigma$ 's head moves on a regular spanning subgraph of  $G$ , of  $m$  edges, we have that the expected meeting time of  $S$  (or  $R$ ) and  $\Sigma$  cannot be less than  $\frac{(n-1)^2}{2m}$ .*

**Corollary 5.** *Since  $m = \Theta(n)$  we get a  $\Theta(n)$  lower bound for the expected communication time. In that sense, our protocol's expected communication time is optimal when the support size is  $\Theta(n)$ .*

## 6 Extensions of Our Work

First of all we notice that our work does not assume any particular motion of hosts not in  $\Sigma$  (other than that we are in non-hostile networks). We pose as an open problem the notion of “capture” of  $S$  (or  $R$ ) in hostile networks. We also remark that any assumption on motions of hosts  $s \notin \Sigma$  will lead to much better upper bounds on the communication time. We plan to investigate the case of varying transmission ranges. We also pose as an open problem the proof of correctness and the efficiency analysis of the proposed alternative implementations, and especially the analytic comparison of the “snake” and the “runners” approach performance. Finally, it is interesting to comparatively study the performance of our approach versus other routing protocols (such as TORA, AODV, LAR) through experiments.

## References

1. M. Adler and C. Scheideler: Efficient Communication Strategies for Ad-Hoc Wireless Networks. In Proc. 10th Annual Symposium on Parallel Algorithms and Architectures (SPAA'98)(1998).
2. D. Aldous and J. Fill: *Reversible Markov Chains and Random Walks on Graphs*. Unpublished manuscript. <http://stat-www.berkeley.edu/users/aldous/book.html> (1999).
3. G. Brightwell, P. Winkler: Maximum Hitting Times for Random Walks on Graphs. Journal of Random Structures and Algorithms, 1:263-276, 1990 (1990).

4. I. Chatzigiannakis, S. Nikolettseas, and P. Spirakis: Analysis and Experimental Evaluation of an Innovative and Efficient Routing Approach for Ad-hoc Mobile Networks. In Proc. 4th Annual Workshop on Algorithmic Engineering (WAE'00)(2000).
5. I. Chatzigiannakis, S. Nikolettseas, and P. Spirakis: An Efficient Routing Protocol for Hierarchical Ad-Hoc Mobile Networks In Proc. 1st International Workshop on Parallel and Distributed Computing Issues in Wireless Networks and Mobile Computing, 15th Annual International Parallel & Distributed Processing Symposium (IPDPS'01)(2001).
6. I. Chatzigiannakis, S. Nikolettseas, and P. Spirakis: An Efficient Communication Strategy for Ad-hoc Mobile Networks. As Brief Announcement in Proc. 20th Annual Symposium on Principles of Distributed Computing, (PODC'01)(2001).
7. I. Chatzigiannakis, S. Nikolettseas, and P. Spirakis: On the Average and Worst-case Efficiency of Some New Distributed Communication and Control Algorithms for Ad-hoc Mobile Networks. Invited Paper in Proc. 1st International Workshop on Principles of Mobile Computing, (POMC'01)(2001).
8. I. Chatzigiannakis, S. Nikolettseas, N. Paspalis, P. Spirakis and C. Zaroliagis: An Experimental Study of Basic Communication for Ad-hoc Mobile Networks. In Proc. 5th Annual Workshop on Algorithmic Engineering (WAE'01)(2001).
9. I. Chatzigiannakis, S. Nikolettseas, and P. Spirakis: An Efficient Communication Strategy for Ad-hoc Mobile Networks (full paper). CTI Technical Report, July 2001, 2001. <http://faethon.cti.gr/adhoc/disc01.html>
10. R. Durrett: *Probability: Theory and Examples*. Wadsworth. (1991).
11. K. P. Hatzis, G. P. Pentaris, P. G. Spirakis, V. T. Tampakas and R. B. Tan: Fundamental Control Algorithms in Mobile Networks. In Proc. 11th Annual Symposium on Parallel Algorithms and Architectures (SPAA'99) (1999)
12. T. Imielinski and H. F. Korth: *Mobile Computing*. Kluwer Academic Publishers. (1996).
13. Y. Ko and N. H. Vaidya: Location-Aided Routing (LAR) in Mobile Ad-hoc Networks. In Proc. 4th Annual ACM/IEEE International Conference on Mobile Computing (MOBICOM'98) (1998).
14. Q. Li and D. Rus: Sending Messages to Mobile Users in Disconnected Ad-hoc Wireless Networks. In Proc. 6th Annual ACM/IEEE International Conference on Mobile Computing (MOBICOM'00) (2000).
15. V. D. Park and M. S. Corson: Temporally-ordered routing algorithms (TORA) version 1 functional specification. IETF, Internet Draft, draft-ietf-manet-tora-spec-02.txt, Oct. 1999. (1999).
16. C. E. Perkins and E. M. Royer: Ad-hoc on demand distance vector (AODV) routing. IETF, Internet Draft, draft-ietf-manet-aodv-04.txt (IETF'99). (1999).
17. J. E. Walter, J.L. Welch and N. M. Amato: Distributed Reconfiguration of Metamorphic Robot Chains In Proc. 19th Annual Symposium on Principles of Distributed Computing (PODC'00) (2000).
18. Y. Zhang and W. Lee: Intrusion Detection in Wireless Ad-hoc Networks. In Proc. 6th Annual ACM/IEEE International Conference on Mobile Computing (MOBICOM'00) (2000).

# A Pragmatic Implementation of Non-blocking Linked-Lists

Timothy L. Harris

University of Cambridge Computer Laboratory,  
Cambridge, UK, [tim.harris@cl.cam.ac.uk](mailto:tim.harris@cl.cam.ac.uk)

**Abstract.** We present a new non-blocking implementation of concurrent linked-lists supporting linearizable insertion and deletion operations. The new algorithm provides substantial benefits over previous schemes: it is conceptually simpler and our prototype operates substantially faster.

## 1 Introduction

It is becoming evident that non-blocking algorithms can deliver significant benefits to parallel systems [MP91, LaM94, GC96, ABP98, Gre99]. Such algorithms use low-level atomic primitives such as compare-and-swap – through careful design and by eschewing the use of locks it is possible to build systems which scale to highly-parallel environments and which are resilient to scheduling decisions.

Linked-lists are one of the most basic data structures used in program design, and so a simple and effective non-blocking linked-list implementation could serve as the basis for many data structures. This paper presents a novel implementation of linked-lists which is non-blocking, linearizable and which is based on the compare-and-swap (CAS) operation found on contemporary processors.

Section 5 sketches a proof of correctness, describes the use of model-checking to perform exhaustive verification within a limited application domain and also describes empirical tests performed on execution traces from an actual implementation.

In Sect. 6 we compare the performance of the new algorithm against that of a lock-based implementation and against an existing non-blocking algorithm. Compared with these other thread-safe algorithms, ours provides the best performance on each of three simulated workloads and for every level of concurrency.

## 2 Overview

In this section we present an overview of our algorithm and the difficulty in implementing non-blocking linked-lists. As a running example consider an ordered list containing the integers 10 and 30 along with sentinel *head* and *tail* nodes:



Such a data structure may comprise cells containing two fields: a **key** field used to store the element and a **next** field to contain a reference to the next cell in the list.

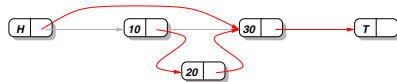
Insertion is straightforward: a new list cell is created (*below, left*) and then introduced using single CAS operation on the **next** field of the proposed predecessor (*below, right*).



In this case the atomicity of the CAS ensures the the nodes either side of the insertion have remained adjacent. This simple guarantee is insufficient for deletions within the list. Suppose that we wish to remove the value 10. An obvious way of excising this node would be to perform a CAS that swings the reference from the head so that the node containing 30 becomes the first in the list:



Although this CAS ensures that the node 10 was still at the start of the list it cannot ensure that no additional nodes were introduced between the 10 node and the 30 node. If this deletion took place concurrently with the previous insertion then that new node would be lost:



The single CAS could neither detect nor prevent changes between 10 and 30 once the deletion procedure had selected 30. Our proposed solution – and indeed the crux of the algorithms presented here – is to use two separate CAS operations in place of that single one. The first of these is used to *mark* the **next** field of the deleted node in some way (*below, left*), whereas the second is used to excise the node (*below, right*):



We say that a node is *logically deleted* after the first stage and that it is *physically deleted* after the second. A marked field may still be traversed but takes a numerically distinct value from its previous unmarked state; the structure of the list is retained while signalling concurrent insertions to avoid introducing new nodes immediately after those that are logically deleted. In our example the concurrent insertion of 20 would observe 10 to be logically deleted and would attempt to physically delete it before re-trying the insertion.

### 3 Related Work

Generalized non-blocking implementations based on CAS were presented by Herlihy [Her91, Her93]. However, linked-lists based on this general scheme are highly centralized and suffer poor performance because they essentially use CAS to change a shared global pointer from one version of the structure to the next.

Valois was the first to present an effective CAS-based non-blocking implementation of linked-lists [Val95]. Although highly distributed, his implementation is very involved. The list is held with *auxilliary cells* between adjacent pairs of ordinary cells. Auxilliary exist to provide an extra level of indirection so that a cell may be removed by joining together the auxilliary cells adjacent to it. Valois' algorithm exposes a more general and lower level interface than we do here; he provides explicit *cursors* to identify cells in the list and operations to insert or delete nodes at those points.

The originally-published algorithm contained a number of errors relating to how reference-counted storage was managed. One has been reported previously and others were identified when implementing Valois' algorithm for comparison in this paper [MS95, Val01].

To overcome the complexity of building linearizable lock-free linked-lists using CAS, Greenwald suggested a stronger double-compare-and-swap (DCAS) primitive that atomically updates two storage locations after confirming that they both contain required values [Gre99]. DCAS is not available on today's multi-processor architectures. However, it does admit a simple linearizable linked-list algorithm: insertions proceed as described in Sect. 2 and deletions by atomic updates to the `next` field of the cell being removed as well as that of its predecessor. Greenwald's work was an extension of earlier non-linearizable DCAS-based linked-list algorithms due to Massalin and Pu [MP91].

### 4 Algorithms

In this section we present our new algorithm in pseudo-code modeled on C++ and designed for execution on a conventional shared-memory multi-processor system supporting *read*, *write* and atomic *compare-and-swap* operations. We assume that the operations defined here are the only means of accessing linked list objects. Each processor executes a sequence of these operations, defining a *history* of invocations/responses and inducing a *real-time* order between them. We say that an operation *A* *precedes* *B* if the response to *A* occurs before the invocation of *B* and that operations are *concurrent* if they have no real-time ordering.

A *sequential* history is one in which each invocation is followed immediately by its corresponding response. Our basic correctness requirement is linearizability which requires that (a) the responses received in every concurrent history are equivalent to those of some legal sequential history of the same requests and (b) the ordering of operations within the sequential history is consistent with the real-time order [HW90]. Linearizability means that operations appear



```

class List<KeyType> {
    Node<KeyType> *head;
    Node<KeyType> *tail;

    List() {
        head = new Node<KeyType> ();
        tail = new Node<KeyType> ();
        head.next = tail;
    }
}

class Node<KeyType> {
    KeyType key;
    Node *next;

    Node (KeyType key) {
        this.key = key;
    }
}

```

**Fig. 1.** An instance of the List class contains two fields which identify the head and the tail. Instances of Node contain two fields identifying the key and successor of the node.

```

public boolean List::insert (KeyType key) {
    Node *new_node = new Node(key);
    Node *right_node, *left_node;

    do {
        right_node = search (key, &left_node);
        if ((right_node != tail) && (right_node.key == key)) /*T1*/
            return false;
        new_node.next = right_node;
        if (CAS (&left_node.next, right_node, new_node)) /*C2*/
            return true;
    } while (true); /*B3*/
}

```

**Fig. 2.** The List::insert method attempts to insert a new node with the supplied key.

to take effect atomically at some point between their invocation and response. Our implementation is additionally *non-blocking*, meaning that some operation will complete in a finite number of steps, even if other operations halt.

We write `CAS(addr,o,n)` for a CAS operation that atomically compares the contents of `addr` against the old value `o` and – if they match – writes `n` to that location. CAS returns a boolean indicating whether this update took place. Our design was guided by the assumption that a CAS operation is slower to execute than a write which in turn is slower than a read.

## 4.1 Implementing Sets

Initially we will consider a set object supporting three operations: `Insert( $k$ )`, `Delete( $k$ )`, `Find( $k$ )`. Each parameter  $k$  is drawn from a set of totally-ordered keys. The result of an Insert, a Delete or a Find is a boolean indicating success or failure. The set is represented by an instance of `List` which contains a singly-linked list of instances of `Node`. As sketched in Sect. 2 these are held in ascending order with sentinel head and tail nodes.

```

public boolean List::delete (KeyType search_key) {
    Node *right_node, *right_node_next, *left_node;

    do {
        right_node = search (search_key, &left_node);
        if ((right_node == tail) || (right_node.key != search_key)) /*T1*/
            return false;
        right_node_next = right_node.next;
        if (!is_marked_reference(right_node_next))
            if (CAS (&(right_node.next), /*C3*/
                    right_node_next, get_marked_reference (right_node_next)))
                break;
    } while (true); /*B4*/
    if (!CAS (&(left_node.next), right_node, right_node_next)) /*C4*/
        right_node = search (right_node.key, &left_node);
    return true;
}

```

**Fig. 3.** The `List::delete` method attempts to remove a node containing the supplied key.

```

public boolean List::find (KeyType search_key) {
    Node *right_node, *left_node;

    right_node = search (search_key, &left_node);
    if ((right_node == tail) ||
        (right_node.key != search_key))
        return false;
    else
        return true;
}

```

**Fig. 4.** The `List::find` method tests whether the list contains a node with the supplied key.

The reference contained in the next field of a node may be in one of two states: marked or unmarked. A node is marked if and only if its next field is marked. Marked references are distinct from normal references but still allow the referred-to node to be determined – for example they may be indicated by an otherwise-unused low-order bit in each reference. Intuitively a marked node is one which should be ignored because some process is deleting it. The function `is_marked_reference(r)` returns `true` if and only if `r` is a marked reference. Similarly `get_marked_reference(r)` and `get_unmarked_reference(r)` convert between marked and unmarked references.

The concurrent implementation comprises four methods (Fig. 2.45). The first three, `List::insert`, `List::delete` and `List::find` implement the Insert, Delete and Find operations respectively. The fourth, `List::search`, is used during each of these operations. It takes a search key and returns references to two nodes called the *left node* and *right node* for that key. The method ensures that these nodes satisfy a number of conditions. Firstly, the key of the left node must be less than the search key and the key of the right node must be greater than

```

private Node *List::search (KeyType search_key, Node **left_node) {
    Node *left_node_next, *right_node;

search_again:
    do {
        Node *t = head;
        Node *t_next = head.next;

        /* 1: Find left_node and right_node */
        do {
            if (!is_marked_reference(t_next)) {
                (*left_node) = t;
                left_node_next = t_next;
            }
            t = get_unmarked_reference(t_next);
            if (t == tail) break;
            t_next = t.next;
        } while (is_marked_reference(t_next) || (t.key < search_key)); /*B1*/
        right_node = t;

        /* 2: Check nodes are adjacent */
        if (left_node_next == right_node)
            if ((right_node != tail) && is_marked_reference(right_node.next))
                goto search_again; /*G1*/
            else
                return right_node; /*R1*/

        /* 3: Remove one or more marked nodes */
        if (CAS (&(left_node.next), left_node_next, right_node)) /*C1*/
            if ((right_node != tail) && is_marked_reference(right_node.next))
                goto search_again; /*G2*/
            else
                return right_node; /*R2*/
        } while (true); /*B2*/
    }
}

```

**Fig. 5.** The `List::search` operation finds the left and right nodes for a particular search key.

or equal to the search key. Secondly, both nodes must be unmarked. Finally, the right node must be the immediate successor of the left node. This last condition requires the search operation to remove marked nodes from the list so that the left and right nodes are adjacent. As we will show the `List::search` method is implemented so that these conditions are satisfied concurrently at some point between the method's invocation and its completion.

`List::search` is divided into three sections. The first section iterates along the list to find the first unmarked node with a key greater than or equal to the search key. This is the right node. The left node preliminarily refers to the previous unmarked node that was found. The second stage examines these nodes. If `left_node` is the immediate predecessor of `right_node` then `List::search` returns. Otherwise, the third stage uses a CAS operation to remove marked nodes between `left_node` and `right_node`.

`List::insert` uses `List::search` to locate the pair of nodes between which the new node is to be inserted. The update itself takes place with a single CAS operation (C2) which swings the reference in `left_node.next` from `right_node` to the new node.

`List::delete` uses `List::search` to locate the node to delete and then uses a two-stage process to perform the deletion. Firstly, the node is logically deleted by marking the reference contained in `right_node.next` (C3). Secondly, the node is physically deleted. This may be performed directly (C4) or within a separate invocation of search.

The `List::find` method is shown in Fig. 4. It invokes `List::search` and examines the resulting right node.

## 5 Correctness

In this section we describe three approaches taken to checking the correctness of the algorithms presented here. Section 5.1 outlines a proof of linearizability and progress, Sect. 5.2 describes the exhaustive testing of some cases through model checking and Sect. 5.3 describes a method we used for examining traces from particular program runs.

### 5.1 Proof Sketch

We will take a fairly direct approach to outlining the linearizability of the operations by identifying particular instants during their execution at which the complete operation appears to occur atomically.

**Conditions Maintained by Search.** Our argument relies on the conditions identified in Sect. 4.1 which the implementation of `List::search` guarantees hold at some point during its invocation. For the ordering constraints, note that when right node is initialized the preceding loop ensured that `search_key`  $\leq$  `right_node.key`. Similarly `left_node.key`  $<$  `search_key` because otherwise the loop would have terminated earlier.

For the adjacency condition and the mark state of the left node we must separately consider each return path. If `List::search` returns at R1 then the test guarding the return statement ensures that right node was the immediate successor of the left node when the `next` field of that node was read into the local variable `t_next`. The same value of `t_next` is found to be unmarked before initializing `left_node`. If `List::search` returns at R2 then C1 establishes the required conditions.

For the mark state of the right node, observe that both return paths confirm that the right node is unmarked after the point at which the first three conditions must be true. Nodes never become unmarked and so we may deduce that the right node was unmarked at that earlier point.

**Linearization points.** Let  $op_{i,m}$  be the  $m^{\text{th}}$  operation performed by processor  $i$  and let  $d_{i,m}$  be the final real-time at which the `List::search` post-conditions are satisfied during its execution. These  $d_{i,m}$  identify the times at which the outcome of the operations become inevitable and we shall take the ordering between them to define the linearized order of `Find(k)` operations or unsuccessful updates. For a successful find at  $d_{i,m}$  the right node was unmarked and contained the search key. For an unsuccessful insertion it exhibits a node with a matching key. For an unsuccessful deletion or find it exhibits the left and right nodes which, respectively, have keys strictly less-than and strictly greater-than the search key.

Furthermore let  $u_{i,m}$  be the real-time at which the update `C2` inserts a node or `C3` logically deletes a node. We shall take  $u_{i,m}$  as the linearization points for such successful updates. In the case of a successful insertion the CAS at  $u_{i,m}$  ensures that the left node is still unmarked and that the right node is still its successor. For a successful deletion the CAS at  $u_{i,m}$  serves two purposes. Firstly, it ensures that the right node is still unmarked immediately before the update (that is, it has not been logically deleted by a preceding successful deletion). Secondly, the update itself marks the right node and therefore makes the deletion visible to other processors.

**Progress.** We will show that the concurrent implementation is non-blocking. We will show that each successful insertion causes exactly one update, that each successful deletion causes at most two updates and that unsuccessful operations do not cause any updates.

The CAS instructions `C1` and `C4` each succeed only by unlinking marked nodes from the list. Therefore the number of times that these CAS instructions succeed is bounded above by the number of nodes that have been marked. Exactly one node is marked during each successful deletion (`C3`) and therefore at most one update may be performed by `C1` or `C4` for each successful deletion. The remaining CAS instructions (`C2` and `C3`) occur respectively exactly once on the return paths from successful insertions and deletions.

Since there are no recursive or mutually-recursive method definitions consider each backward branch in turn:

- Each time `B1` is taken the local variable `t` is advanced once node down the list. The list is always contains the unmarked tail node and the nodes visited have successively strictly larger keys.
- Each time `B2` is taken the CAS at `C1` has failed and therefore the value of `left_node.next`  $\neq$  `left_node.next`. The value of the field must have been modified since it was read during the loop ending at `B1`. Modifications are only made by successful CAS instructions and each operation causes at most two successful CAS instructions.
- Each time `B3` or `B4` is taken the CAS at `C2` or `C4` has failed. As before, the value held in that location must have been modified since it was read in `List::search` and at most two such updates may occur for each operation.

- Each time G1 or G2 is taken then a node which was previously unmarked has been marked by another processor. As before, at most two updates may occur for each operation.

## 5.2 Model Checking

The dSPIN model checker was used to exhaustively verify the operations for certain problem domains. dSPIN is an extension of the SPIN model checker with adds support for pointers, storage management, function calls and local scopes [Hol97][S99]. This made it more suitable than SPIN for a natural representation of these algorithms.

The modeled state contains two representations of the set: one comprises a linked list of cells whereas the other is summarized as a bit vector. The linked list is updated as proposed here using atomic `d_step` instructions to implement CAS. The bit vector is checked or updated using further `d_steps` at the proposed linearization points.

The model was parameterized according to the number of concurrent threads, the number of operations that each would attempt and the range of key values that could be used. The two largest configurations we could practicably test were with four threads, each performing one operation with three potential keys and with two threads each performing two operations with four potential keys.

## 5.3 Practical Testing

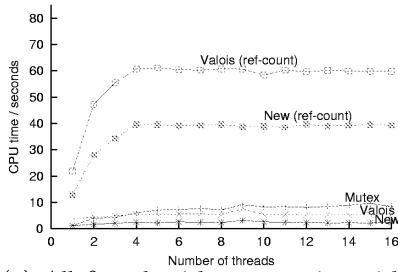
The linearizability of the operations has also been tested pragmatically. Although such tests cannot provide the assurances of formal methods they are nonetheless important because they avoid the need to make simplifying assumptions for tractability. In particular, the use of relaxed memory models means that the operations supported by a conventional shared memory machine are not linearizable; a direct implementation is likely to fail without further memory barrier instructions.

It is not generally possible to record actual timestamp values for arbitrary operations within an running process. Instead, we surrounded the code executed at each linearization point with further instructions to record coherent per-processor cycle counts.

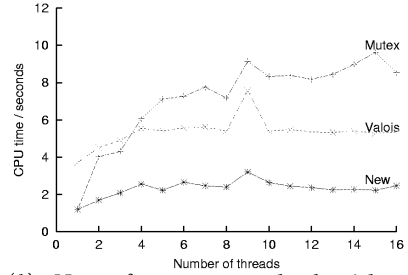
The resulting intervals were recorded to an in-memory log which was then replayed sequentially in timestamp order. The results thus obtained were compared with those from the concurrent execution. The replay program contains simple heuristics to deal with overlapping intervals. If these cannot determine a consistent linearized order then the replay program reports unresolved inconsistencies for manual inspection and re-ordering.

## 6 Results

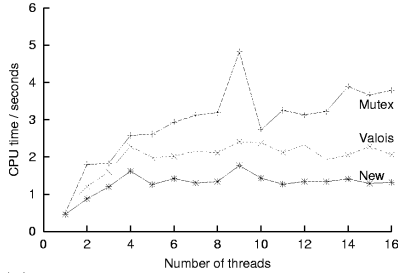
The algorithm described in Sect. 4 has been implemented in a combination of C and SPARC V9 assembly language. We evaluated its performance on an E450



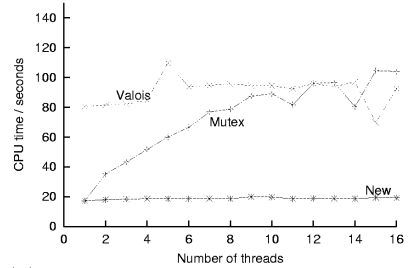
(a) All five algorithms operating with keys in the range 0...255.



(b) Non-reference-counted algorithms with keys 0...255.



(c) Non-reference-counted algorithms all operating on key 0.



(d) Non-reference-counted algorithms on keys 0...8191.

**Fig. 6.** CPU time (user+system) accounted to the benchmark application for each algorithm on a variety of workloads. In each case the x-axis shows the number of concurrent threads.

server running Solaris 8 and fitted with four 400MHz SPARC V9 processors and 4GB physical memory. It is worth emphasising that the code in Fig. 11-4 is intended merely as pseudo-code and does not reflect an optimised (or even necessarily correct) implementation. Processors may require additional memory barriers – for example between initializing the fields of a new node and introducing it into the list, or between the CAS that logically deletes a node and the CAS that physically deletes it.

The test application compared our implementation against Valois' lock-free algorithm and against a straightforward one in which the list is protected by a mutual exclusion lock<sup>1</sup>. Both lock-free algorithms were evaluated with and without reference-counting. All list cells were allocated ahead of time so that the performance of particular memory allocation functions was not included in the results. The code to manipulate reference-counts is based on Valois' as modified

<sup>1</sup> This comparison against a lock-based algorithm is somewhat unfair: the simplified programming model there makes it straightforward to implement a more efficient data structure such as a tree or skiplist.

by Michael and Scott with the exception that reference counts are recursively decremented when a cell is freed.

We generated a workload of insertion and deletion operations by randomly choosing keys uniformly distributed within a particular range, selecting equiprobably between insertions and deletions. We used per-thread linear congruential random number generators with the same parameters as the `lrand48` function from the Solaris 8 `libc` library. Seeds were chosen to give non-overlapping series.

The test harness was parameterized on the algorithm to use, the number of concurrent threads to operate and the range of keys that might be inserted or deleted. In each case every thread performed 1 000 000 operations. Figure 6 shows the CPU accounted to the process as a whole for each of the algorithms tested on a variety of workloads.

It is immediately apparent that our algorithm performs notably better for every experiment using more than one thread. In the case of single-threaded execution it outperforms Valois' algorithm in these tests and its performance equals that the lock-based implementation. The relative performance compared with Valois' algorithm is not surprising: we avoid the need to create, traverse and excise auxilliary nodes.

In addition to the workloads presented in those graphs we also tested configurations with larger ranges of keys, or where the list was initially 'primed' with a long sequence of nodes that would never be deleted. In each case this increased the total number of nodes in the list and thereby added to the cost of retrying operations when CAS instructions fail. One fear was that the lock-free algorithms would start to perform poorly because of the potential for multiple retries. We studied workloads up to lists of 65 536 elements and were unable to find any configuration for which the algorithms based on mutual-exclusion give the best performance. We suspect that although each retry becomes more costly, the likelihood of retries decreases as the rate of conflicting updates falls.

Figure 6a shows the performance of reference-counted implementations. The CPU requirements of Valois' algorithm are degraded by a factor of 5 in the single-threaded case, rising to over 11 for sixteen threads. Similarly, the CPU time required by our algorithms is degraded by a factor of 10 rising to over 15. In each case this is a consequence of need to manipulate reference-counts (using CAS operations) at each stage during a list's traversal. Valois reports that he had originally intended to assume the use of a tracing garbage collector [Val01].

The performance of reference-count manipulation is hampered because the SPARC processor does not provide atomic fetch-and-add. However, measurements taken on a dual-processor Intel x86 machine (with that facility [PPt96]) suggest that the degradation is low when compared with the overall costs seen here. When the reference counts lie on separate lines in the L1 data cache then updates implemented through CAS are 10% slower than those using fetch-and-add. This rises to a factor of 2 degradation when the two processors attempt to update the same address.

Of course, our results are optimistic in that they do not consider the cost of performing GC. However, as Jones and Lins write, if the size of the active data



structure is fixed then the cost of copying collectors may be reduced arbitrarily at the expense of the total heap size [JL96]. More practically, they report that overall costs of around 10-20% are typical in modern well-implemented systems.

We examined a further approach to storage reclamation based on the deferred freeing of nodes. In this scheme each node contains an additional field through which it can be linked onto a to-be-freed list when it is excised from the main list. Each thread takes a snapshot of a global timer as its *current time* before starting each operation. Entries are removed from a to-be-freed list when the time of their excision precedes the minimum current time of any thread: at that point no thread can still have a reference to the node held in any of its local variables.

Our implementation allocates a pair of to-be-freed lists for each thread. These are termed the *old list* and the *new list* and are held along with a separate per-thread timer snapshot that is more recent than the excision time of any element of the old list. When the minimum current time exceeds the snapshot then the entire contents of the old list are freed and the elements of the new list are moved to the old list.

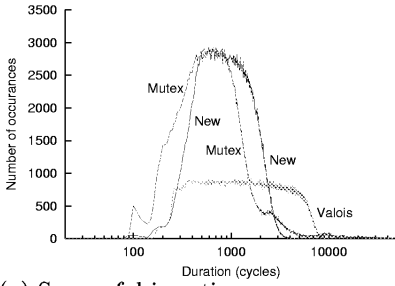
This deferred freeing scheme introduces two principal overheads when compared with the used of garbage collection. Firstly, a CAS operation is needed to place nodes on a to-be-freed list – in our implementation this increased the CPU requirements by 15% compared with the results from Fig. 6a operating with 16 threads. The second overhead is the cost of removing elements from the to-be-freed lists and establishing when it is safe to do so. This was a further 1% when performed every 1000 operations and 5% every 100, rising to 52% if performed after every operation.

Figure 7 presents a further analysis of the run-time performance of the three non-reference-counted algorithms, showing the distribution of execution-times for four different kinds of operation. These results were gathered when 8 concurrent threads performing insertions and deletions of keys in the range  $0 \dots 255$ . In the case of successful operations the lock-based implementation is able to achieve lower execution times than either lock-free scheme. However, it is also occasionally prone to much longer execution times which explain the higher mean execution time suggested by Fig. 6.

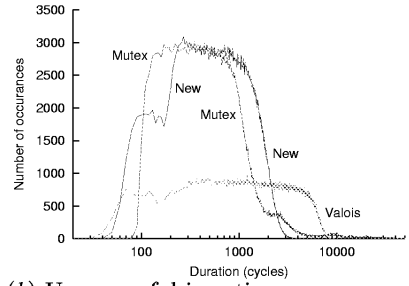
The situation is somewhat different for unsuccessful operations in that both lock-free algorithms obtain some execution times which are lower than those of the lock-based implementation – recall that unsuccessful operations may occur without requiring any CAS operations or other updates to the data structure.

## 7 Delete Greater-Than-or-Equal

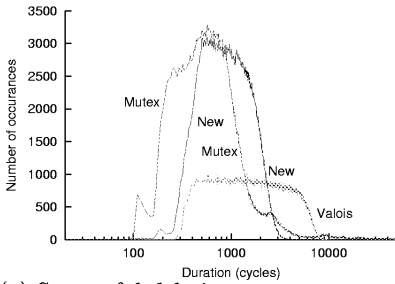
Now consider the problem of implementing a further operation of the form  $\text{DeleteGE}(k)$  which returns and removes the smallest item that is greater than or equal to  $k$ . It is tempting to implement this by modifying `List::delete` so that the test T1 does not fail if the key of the right node is greater than the search key.



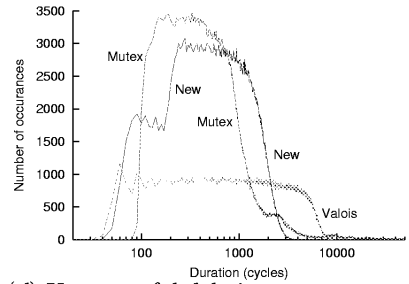
(a) Successful insertions



(b) Unsuccessful insertions



(c) Successful deletions



(d) Unsuccessful deletions

**Fig. 7.** Operation-time distributions. Each graph shows execution times (in processor cycles) on the x-axis and numbers of occurrences on the y-axis.

Unfortunately this implementation is not linearizable. Suppose that three insertion operations are executed in sequence: `Insert(20)`, `Insert(15)`, `Insert(20)`. The first two succeed and the third must fail because 20 is already in the set. However, consider a concurrent `DeleteGE(10)` operation, attempting to delete any node with a key greater than or equal to 10. Concurrent execution may proceed as follows:

- `List::deleteGE` invokes `List::search` immediately after the first insertion of 20. It takes the head of the list as the left node and the node containing 20 as the right.
- The successful insertion of 15 occurs.
- The unsuccessful insertion of 20 occurs, observing 15 as the key of its left node and 20 as the key of its right node.
- `List::deleteGE(10)` completes after logically deleting the node containing 20.

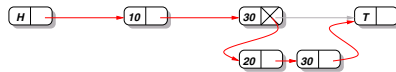
We must order this `DeleteGE(10)` operation such that its result of 20 would be obtained by a sequential execution. This requires it to be placed before the insertion of 15 because otherwise the key 15 should have been returned in preference to 20. However, we must also linearize the deletion after the failed insertion

of 20 because otherwise that insertion would have succeeded. These constraints are irreconcilable.

Intuitively the problem is that at the execution of `C3` the right node need not be the immediate successor of the left node. This was acceptable when considering the basic `Delete( $k$ )` operation because we were only concerned with concurrent updates affecting nodes with the same key. Such an update must have marked the right node and so `C3` would have failed. In contrast, during the execution of `List::deleteGE`, we must be concerned with updates to any nodes whose keys are greater than or equal to the search key.

We can address this by retaining the implementation of `List::deleteGE` but changing `List::insert` in such a way that `C3` must fail whenever a new node may have been inserted between the left and right nodes. This would mean that, whenever `C3` succeeds, the key of the right node must still be the smallest key that is greater than or equal to the search key.

This is achieved by using a single CAS operation to (a) introduce a pair of new nodes, one that contains the value being inserted and another that duplicates the right node and (b) mark the original right node:



Such a CAS conceptually has two effects. Firstly, it introduces the new node into the list: beforehand the `next` field of the successor is unmarked and therefore the right node must still be the successor of the left node. Secondly, by marking the contents of that `next` field, the CAS will cause any concurrent `List::deleteGE` with the same right node to fail. Note that the key of the now-marked right node is not in the correct order. However, the existing implementations of `List::search`, `List::delete`, `List::find` and `List::deleteGE` are written so that they do not rely on the correct ordering of marked nodes.

## 8 Conclusion

This paper has presented a new non-blocking implementation of linked lists. We believe that the algorithms presented here are linearizable. They have also been implemented and we have shown that their measured performance improves both on previously published non-blocking data structures and also on a lock-based implementation.

**Acknowledgments.** The work described in this paper was carried out during an internship with the Java Technology Research Group at Sun Labs. The design presented here is the result of much fruitful discussion with Dave Detlefs, Christine Flood, Alex Garthwaite, Steve Heller, Nir Shavit and Guy Steele. The implementation and evaluation have similarly benefitted from feedback from Mike Burrows, Keir Fraser, Steven Hand, Mark Moir and John Valois.

## References

- [ABP98] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the 10th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 119–129, Puerto Vallarta, Mexico, June 28–July 2, 1998. SIGACT/SIGARCH.
- [GC96] Michael Greenwald and David Cheriton. The synergy between non-blocking synchronization and operating system structure. In *USENIX, editor, 2nd Symposium on Operating Systems Design and Implementation (OSDI '96), October 28–31, 1996. Seattle, WA*, pages 123–136, Berkeley, CA, USA, October 1996. USENIX.
- [Gre99] M Greenwald. *Non-blocking synchronization and system design*. PhD thesis, Stanford University, August 1999. Technical report STAN-CS-TR-99-1624.
- [Her91] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.
- [Her93] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.
- [Hol97] Gerard J Holzmman. The moel checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
- [HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.
- [IS99] Radu Iosif and Riccardo Sisto. dSPIN: A dynamic extension of SPIN. In *Proc. of the 6th International SPIN Workshop*, volume 1680 of *LNCS*, pages 261–276. Springer-Verlag, September 1999.
- [JL96] Richard Jones and Rafael Lins. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, July 1996.
- [LaM94] Anthony LaMarca. A performance evaluation of lock-free synchronization protocols. In *Proceedings of the Thirteenth Symposium on Principles of Distributed Computing*, pages 130–140, 1994.
- [MP91] Henry Massalin and Calton Pu. A lock-free multiprocessor OS kernel. Technical Report CUCS-005-91, Columbia University, 1991.
- [MS95] Maged M. Michael and Michael L. Scott. Correction of a memory management method for lock-free data structures. Technical Report TR599, University of Rochester, Computer Science Department, December 1995.
- [PPr96] *Pentium Pro Family Developer's Manual, volume 2, programmer's reference manual*. Intel Corporation, 1996. Reference number 242691-001.
- [Val95] John D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 214–222, Ottawa, Ontario, Canada, 2–23 August 1995.
- [Val01] John D Valois. Personal communication. March 2001.

# Stabilizing Replicated Search Trees

Ted Herman<sup>1\*</sup> and Toshimitsu Masuzawa<sup>2\*\*</sup>

<sup>1</sup> University of Iowa

herman@cs.uiowa.edu

<sup>2</sup> Osaka University, Japan

masuzawa@ics.es.osaka-u.ac.jp

**Abstract.** This paper proposes a weakening of stabilization suited to the model of objects updated and queried by operations (method invocations). The paper's main result is a construction for replicated search trees in a message-passing, synchronous distributed system. Any sequence of  $O(d)$  operations brings all trees to legitimate and consistent states, where  $d$  is the maximum number of items accessible in the set of search trees at the initial state.

## 1 Introduction

The discovery of many fundamental concepts in distributed computing can be traced back to research motivated by distributed databases, and especially replicated data. Replication is useful for fault tolerance and also to enhance availability. Most often, fault tolerance is taken to mean either masking the effects of failures by some form of redundancy or by restoring computation from backup or checkpoint states. In the former case, the tolerance depends on limitations of the failure with respect to the amount of redundancy; in the latter, tolerance depends on stable storage unaffected by failures. Notice that the idea of restoring from a backup copy can interfere with availability because new service requests may have to wait until the restoration is complete. In contrast to these techniques of redundancy or backup/restore, the paradigm of (self-) stabilization is *forward* error recovery, which is to say that a system makes do with a faulty state, by applying repairs to data without recourse to backup copies. Further, for stabilization there is no limit on the number of transient failures, so redundancy is not a primary issue. Unfortunately, the model of stabilization does not address two vital availability concerns: first, the behavior of a stabilizing system can be erratic during periods of recovery from a transient failure, and second, there are few investigations of replicated objects in the literature of stabilizing systems.

This paper initiates an effort to combine the concerns of availability with the forward recovery paradigm of stabilization in a distributed system. We chose the problem of a replicated search tree as our case study because a search tree is similar to many indexing structures used in the important application of distributed databases (in fact, our construction uses a 2-3 tree, which is a simple case of a

---

\* Research supported by NSF award CAREER 97-9953.

\*\* This work is supported in part by Japan Society for the Promotion of Science (JSPS), Grant-in-Aid for Scientific Research((c)(2)12680349).

B-tree). The significant properties of the algorithm presented in this paper are: a search tree with find, insert, and delete operations supports  $t$  identical copies of the data structure (one per site); find operations are local to a site to enhance availability; starting from data structures with arbitrary states (replicas that are not identical, data structures with damaged variables), recovery to a legitimate state with identical replicas is guaranteed; and operations can be applied to the search tree even while the global configuration is illegitimate, yet the running time of such operations is reasonable and the response to such operations is guaranteed to be consistent with the operation's effect at all sites. The construction is fully symmetric (not relying on a master copy for the replicas) and thus has other failure tolerances not described in this extended abstract.

**Related Work.** Literature on replicated data, system availability, and fault-tolerant data structures is vast; we cite mainly here the relation of the present research to previous work that is self-stabilizing. Until recently, studies of stabilization did not attend to behavioral constraints during periods of recovery from a fault event (papers [4,8,14] are the exception). Several papers [1,3,6,12] have described methods for limiting the exposure of erroneous system output by adaptively decreasing the length of the period of convergence following a fault. In fact, some of these papers use data replication as a technique, but this data replication is not motivated directly by availability concerns. Nearly all published works in the area use a network model of nodes and links, and the computational task is self-contained (interaction with agents outside the system is seldom mentioned). Few papers [9,10,11] consider an object-centric model, in which a defined set of operations manipulate objects. Papers using an object model are concerned not only with the internal state of the object (which should eventually become legitimate from any initial state), but also with how operations behave and respond during times of instability.

**Organization.** After presenting the model in Section 2 and reviewing stabilization and availability concepts for an individual site in Section 3, we provide a definition of what is an available and stabilizing replicated search tree in Section 4. An overview of the construction is given in Section 4.2, with details appearing in following sections. We use atomic broadcast to ensure replica coherence, and a secondary contribution of this paper is the introduction of a stabilizing atomic broadcast in Section 4.3. Due to a lack of space in this extended abstract, proofs of correctness have been omitted. Section 5 concludes the paper.

## 2 Distributed System Model

We adopt a model of a distributed system based on I/O automata [13] where sites, channels and a distributed system itself are modeled as I/O automata. This extended abstract omits details of the model because of lack of space. We consider a synchronous distributed message-passing system consisting of  $t$  sites. Each site is connected to any other site by a FIFO message channel (one channel for each direction). Sites and FIFO channels are reliable: each site correctly executes its

program and each channel correctly transfers messages in the order they are sent.

The distributed system considered in this paper is a *synchronous* one because we make the following two assumptions. **(A1)** *message delay* between any two sites is bounded by some upper bound  $\delta$  and the bound  $\delta$  is known to every site; **(A2)** each site has a *timer* keeping perfect time and can set the timer to raise an *alarm* at a preset time interval.

A *configuration* of a distributed system consists of the states of all sites and channels. An *execution* of a distributed system is a (possibly infinite) alternative sequence of configurations and *events*  $E = \sigma_0, e_1, \sigma_1, e_2, \sigma_2 \dots$  such that occurrence of event  $e_i$  changes the configuration from  $\sigma_{i-1}$  to  $\sigma_i$ . Notice that an event sending or receiving a message changes the states of a site and a channel connecting to the site.

Since we consider a synchronous distributed system, we sometimes deal with a *timed execution*  $\sigma_0, (e_1, t_1), \sigma_1, (e_2, t_2), \sigma_2 \dots$  where each event  $e_i$  is associated with time  $t_i$  when the event occurs. The timed execution we consider satisfies the following four properties: (i) the times assigned to events are non-decreasing, that is,  $t_{i-1} \leq t_i$  holds for any  $i$ ; (ii) if  $(e, t)$  is an event sending a message  $M$  to a site  $i$ , there exists event  $(e', t')$  receiving  $M$  at site  $i$  such that  $t < t' < t + \delta$  (this implies that the message delay has an upper bound  $\delta$ ); (iii) if  $(e, t)$  is  $i$ 's event setting its timer to  $\tau$ , then there exists  $i$ 's alarm event  $(e', t')$  such that  $t' = t + \tau$  unless  $i$  resets the timer before  $t'$ ; (iv) If an internal or send event  $e$  is applicable at  $\sigma_k$ , then there exists  $(e, t)$  such that  $t = t_k$  where  $t_k$  is the time assigned to the event that changes the configuration to  $\sigma_k$  (this implies that we ignore time for internal and send events, that is, several internal and send events can be executed in an instant).

Since a stabilizing replicated search tree starts with an arbitrary initial configuration, the initial configuration may contain in-transit messages in some FIFO channels. We assume that such messages are received in time  $\delta$  from the initial configuration. In other words, the following holds: if  $(e, t)$  is  $i$ 's event receiving a message  $M$  at time  $t$  ( $\geq \delta$ ), then there exists event  $(e', t')$  sending  $M$  to  $i$  such that  $t' < t$ .

### 3 Site Availability and Stabilization

Each site is the host of one replica of the search tree. Three operations, **insert**, **delete**, and **find**, are defined on the search tree at each site. An **insert**( $x$ ) operation inserts the item (with key)  $x$  and has two possible responses, *ok* (indicating that  $x$  was successfully inserted or already present), or *full* (the **insert** was aborted because the tree is full). A **delete**( $x$ ) operation deletes the item  $x$  from the tree (if it exists) and always responds *ok*. A **find**( $x$ ) operation either returns the item  $x$  in the tree, *empty* (indicating that the tree is empty) or *miss* (the tree is not empty but does not contain  $x$ ).

Because initial configurations can be arbitrary, the internal variables of data structures within a site can be corrupt in the initial state. The implementation

of a search tree presented in [10] satisfies the following conditions even when started from an arbitrary initial state. **(B1)** [*Stabilization*] After some number of operations, a point  $z$  in the history of operations occurs, after which the tree behaves as a usual 2-3 tree: the running time of any operation is  $O(\lg Z)$  where  $Z$  is the size of the tree at the point where the operation runs, and an **insert**( $x$ ) operation definitely responds *full* if  $Z = M$ , and may respond *full* if  $2M/3 \leq Z < M$  where  $M$  is the maximum capacity of the tree. (The implementation in [10] guarantees reaching the point  $z$  of (B1) within  $O(d)$  tree operations where  $d$  is the number of nodes reachable from the root in the 2-3 tree at the initial state.) **(B2)** [*Availability*] Even during convergence to a legitimate state, a successful **insert**( $x$ ) guarantees that the tree contains  $x$  until some **delete**( $x$ ) is applied; if a **find**( $x$ ) operation returns  $x$  even during convergence to a legitimate state, then subsequent **find**( $x$ ) operations do not return *miss* (or *empty*) until some **delete**( $x$ ) is applied. The complete definition of availability in [10] specifies that the data structure offers some reliability and responsiveness guarantees at all points of any operation history: in particular, before the point  $z$  is reached, the running time of any operation is  $O(\lg M)$ , though any **insert**( $x$ ) operation may respond *full* even if  $Z < 2M/3$ .

Although we seldom refer to the availability and stabilization properties of each replica in this extended abstract, they are crucial to our distributed implementation: so long as sufficiently many operations are applied to a replica, that replica's state converges to a legitimate state.

## 4 Distributed Stabilization

### 4.1 Stabilizing Replicated Tree

An important motivation of replicated search trees can be to increase the availability to applications by reducing the latency of **find** operations. Another important motivation is to preserve global consistency. It is well known from studies of cache coherency, that these motivations can conflict. Our choice will be to let **find** operations execute only on local copies of search trees. This choice violates some consistency criteria: a perfectly consistent implementation would serialize **find** operations along with **insert** and **delete** operations, but to do so would not allow the **find** implementation to be entirely local.

Let  $S_i$  ( $1 \leq i \leq t$ ) be a replica of the search tree at site  $i$  (there are  $t$  sites hosting replicas). Every operation on the replicated search tree has an *originating* site. A history of operations on the replicated search tree is a sequence of operations and their responses. Because operations on the replicated search tree are specified in terms of the semantics of site operations, we denote operations to the replicated search tree by **R.find**, **R.insert** and **R.delete**, and denote operations on replica  $S_i$  by **insert** <sub>$i$</sub> , **delete** <sub>$i$</sub>  and **find** <sub>$i$</sub> . In the definition based on I/O automata, **R.find**, **R.insert** and **R.delete** are input events from the outside world, while **insert** <sub>$i$</sub> , **delete** <sub>$i$</sub>  and **find** <sub>$i$</sub>  are internal events at  $S_i$ . Each site also has an output event **respond** <sub>$i$</sub>  to return the response to the outside world. The input events **R.find**, **R.insert** and **R.delete** are not under the sites'



control and, thus, they may occur at any site and at any time. However, we assume that each site can receive a new tree operation from the outside world only after it returns the response to the previous tree operation originating at the site.

To define the contents of a replicated search tree we would like to assume that **R.insert** and **R.delete** operations are linearized in any history (the atomic broadcast defined later enables this assumption). For any point  $\sigma$  in a linearized history of operations on a replicated search tree, the *contents* of the replicated search tree is the set of items  $C_\sigma$  defined by:  $x \in C_\sigma$  iff there is a point  $\sigma'$  previous to  $\sigma$  where a successful **R.insert**( $x$ ) operation is applied at  $\sigma'$  (i.e., the operation responded *ok*) and there is no **R.delete**( $x$ ) operation applied between  $\sigma'$  and  $\sigma$ ; also, the contents of replica  $S_i$  ( $1 \leq i \leq t$ ) at the linearized point  $\sigma$  is defined by a history that replaces each **R.insert**( $x$ ) by **insert** <sub>$i$</sub> ( $x$ ) and each **R.delete**( $x$ ) by **delete** <sub>$i$</sub> ( $x$ ). Thus all replicas have equal contents at each point in the linearized history. Any history of operations on the replicated search tree satisfies: **(C1)** An **R.insert**( $x$ ) operation may respond *full* only if the number  $Z$  of items contained in the replicated search tree satisfies  $K_i \leq Z \leq M_i$  at some replica  $S_i$ , where values  $K_i$  and  $M_i$  are specific to the sequential implementation of  $S_i$ . An **R.insert**( $x$ ) operation definitely responds *full* if  $Z = M_i$  at some replica  $S_i$ . **(C2)** An **R.find**( $x$ ) operation originating at site  $i$  returns *empty* if replica  $S_i$  contains no items. Otherwise, **R.find**( $x$ ) operation returns  $x$  if  $x$  is in  $S_i$ , and returns *miss* if  $x$  is not in  $S_i$ . The above properties imply that the initial content of the replicated search tree and every replica is the empty set. In the following, we assume for simplicity that all sites have identical  $K_i$  and  $M_i$  denoted by  $K$  and  $M$  respectively.

In a *stabilizing* replicated search tree, we make no assumption on its initial configuration; for example, the contents of one replica may differ from that of another. In this case, **R.find**( $x$ ) operations originating at different sites but for the same key  $x$  may respond differently, since the **R.find** operation is an entirely local operation. However, a stabilizing replicated search tree is required to have identical contents in all replicas eventually, thus, some items should be inserted and/or deleted at some replicas in addition to operations invoked by the outside world. To specify operation behavior for stabilization, thus, we use an augmented linearized history of operations. Let  $\mathcal{H}$  be any history of operations on a replicated search tree. Let  $\mathcal{H}_i$  be the projection of  $\mathcal{H}$  for site  $i$ , that is, replace all **R.insert** operations by **insert** <sub>$i$</sub>  operations, all **R.delete** operations by **delete** <sub>$i$</sub>  operations, and remove all **R.find** operations not originating at site  $i$ . A replicated search tree is *stabilizing* if there exists, for each replica  $S_i$  ( $1 \leq i \leq t$ ), a sequence  $P_i$  of at most  $3M$  **insert** <sub>$i$</sub>  and **delete** <sub>$i$</sub>  operations such that  $\mathcal{H}'_i$  satisfying the following conditions (C3) and (C4) can be obtained by shuffling  $P_i$  and  $\mathcal{H}_i$  with the last operation of  $P_i$  occurring within  $O(\max\{|P_i| \mid 1 \leq i \leq t\})$  operations of  $\mathcal{H}'_i$ . (Intuitively,  $P_i$  consists of at most  $M$  **insert** <sub>$i$</sub>  operations for the initial contents of  $S_i$ , and  $M$  **insert** <sub>$i$</sub>  and  $M$  **delete** <sub>$i$</sub>  operations for convergence to the contents identical to all sites.) In the following conditions,  $z_i$  is the last point of  $P_i$  in  $\mathcal{H}'_i$ . **(C3)** After point  $z_i$ , properties (C1) and (C2) hold. Prior to

point  $z_i$ , any  $\text{R.insert}(x)$  operation may respond *full*, even if the contents of every site  $S_i$  is below the threshold value  $K$ . (C4) If any  $\text{insert}_i(x)$  operation has an *ok* response at  $z'_i$  before  $z_i$ , and this  $\text{insert}_i(x)$  operation comes from  $\mathcal{H}_i$ , then no  $\text{delete}_i(x)$  operation from  $P_i$  occurs after  $z'_i$ . This implies that the item  $x$  inserted by a successful  $\text{R.insert}(x)$  operation is guaranteed to be in the replicated search tree until  $\text{R.delete}(x)$  is invoked. Note that (C4) is similar to the site availability requirement (B2).

In the above, we consider all operations are invoked after the initial configuration. In an initial configuration, however, an operation with corrupted control variables may be in progress. Considering our protocol with much care, we can see that it works correctly even when starting with such initial configurations. However, for lack of space, and, for simplicity, we are only concerned with an initial configuration and new operations in this extended abstract.

## 4.2 Overview of Construction

As observed in the previous subsection, all update operations should be executed in the same order at the all sites to keep consistency among the replicas. Our implementation relies on a self-stabilizing atomic broadcast (abbreviated as **ss-ABcast**) of the update operations, which is introduced in this paper. The basic idea to implement the **ss-ABcast** is to provide (synchronous) rounds and to broadcast and deliver messages based on the rounds: each site can broadcast at most one message at each round, and all the messages broadcast at the round are delivered in the same order at all sites within the round. Section 4.3 shows our implementation of the **ss-ABcast**.

To implement a stabilizing replicated search tree, we assume each replica is itself implemented by the stabilizing and available search tree proposed by [10]. However, this is not sufficient to implement a stabilizing replicated search tree if different replicas have different contents. To solve this problem, we require some mechanism for convergence in addition to the mechanism for usual search tree operations. Thus, our implementation of stabilizing replicated search tree repeats two phases, the *tree operation phase* and the *convergence phase*, alternatively. Each phase consists of two rounds of the **ss-ABcast**.

In the tree operation phase, each site receives from the outside world at most one update operation and broadcasts the operation using the **ss-ABcast**. The operations are applied at all sites in the same order (i.e., in the order that they are delivered), and are committed or aborted depending on the responses to the operations in the second round of the phase. Section 4.4 shows more details of the tree operation phase. In the convergence phase, some tree operations are executed to check and correct the contents of each replica so that all replicas should eventually have identical contents. Section 4.5 presents the convergence phase.

Synchronization of phases is needed for our construction to work properly. Since we consider execution starting from any initial configuration, execution may start from the initial configuration where some sites are executing the tree

operation phase but other sites are executing the convergence phase. This inconsistency can be easily detected and corrected by, for instance, returning to the beginning of the tree operation phase (i.e., resetting). Thus, in the rest of this paper, we assume that all sites are synchronized to execute the same phase, and we present each phase independently from the other.

### 4.3 Self-Stabilizing Atomic Broadcast

*Atomic broadcast* [7] guarantees that all messages broadcast by sites are delivered at all sites in the same order. More precisely, atomic broadcast satisfies the following properties: validity, integrity and total order. In the following, **ABcast**( $m$ ) and **Deliver**( $m$ ) are operations that are invoked to broadcast a message  $m$  and to deliver a message  $m$  respectively. **(D1) Validity**: If a site invokes **ABcast**( $m$ ), then every site eventually invokes **Deliver**( $m$ ). **(D2) Integrity**: For any message  $m$ , every site invokes **Deliver**( $m$ ) at most once, and only if some site invoked **ABcast**( $m$ ). **(D3) Total Order**: If a site invokes **Deliver**( $m$ ) before **Deliver**( $m'$ ), then any other site also invokes **Deliver**( $m$ ) before **Deliver**( $m'$ ).

A *self-stabilizing atomic broadcast* (**ss-ABcast**) protocol is a protocol that satisfies the validity, the integrity and the total order after finite time. Our implementation of **ss-ABcast** uses synchronous rounds to fix a set of messages to be delivered in a round. Each site can sequentially broadcast several messages by invoking **ABcast** operations, but we assume that the site invokes **ABcast**( $m_2$ ) only after **Deliver**( $m_1$ ) (if it invoked **ABcast**( $m_1$ ) before).

Figure 1 shows our **ss-ABcast** protocol for each site  $i$  ( $1 \leq i \leq t$ ). The key idea of the protocol is to provide rounds for all sites and it is achieved as follows: To separate rounds, we introduce a *quiet period* between two consecutive rounds where no message is exchanged. If a site receives no message during  $2\delta$  time the site judges the current round ends and the next round begins. Timer  $TA_i$  is used to keep the time of the quiet period. The length  $2\delta$  of the quiet period is chosen as follows: consider two messages  $m$  and  $m'$ , and let  $t$  and  $t'$  (assume  $t < t'$ ) be the times when  $m$  and  $m'$  are broadcast. If  $t' - t < \delta$ , the arrival order of the two messages at a site is unpredictable. It is possible that  $m$  is received before  $m'$  at a site while  $m'$  is received before  $m$  at another site. Our decision is, thus, these messages are delivered at the same round. On the other hand, the difference between arrival times of these messages may be almost  $2\delta$  at a site if  $t' - t$  is almost  $\delta$  and the message delays of  $m$  and  $m'$  are almost 0 and  $\delta$ . Therefore, we choose  $2\delta$  as length of the quiet period.

Since message delay varies from 0 to  $\delta$ , the times when one broadcast message is received may differ by  $\delta$  at distinct sites. Thus, if a site broadcasts a message of the next round immediately on detection of the end of the round, some site cannot necessarily have a quiet period of  $2\delta$  time. This requires additional  $\delta$  time until the site can broadcast the message. (Timer  $TB_i$  is used to keep the time of the additional period.) The following lemma implies that **ss-ABcast** guarantees the properties of the atomic broadcast if we ignore “spurious” initial messages, which are delivered only at the first round.

```

local variables of site  $i$ 
   $TA_i$ :timer; /* a countdown timer for a quiet period. */
   $TB_i$ :timer; /* a countdown timer for starting the next round */
   $rmsg_i$ :array [1.. $t$ ] of message;
    /*  $rmsg_i[j]$  stores a message site  $i$  receives from site  $j$  */

On  $ABcast(m)$  /* On receipt of broadcast request of  $m$  */
  if ( $TA_i$  or  $TB_i$  is running)
    wait until  $alarm(TB_i)$ 
  send  $m$  to all sites; /* including  $i$  */

On receipt of message  $m$  from site  $j$ 
   $rmsg_i[j] := m$ ;
  if  $TB_i$  is running
    set-timer( $TB_i, 0$ ); /* Cancel  $TB_i$  and immediately invoke  $alarm(TB_i)$  */
  set-timer( $TA_i, 2\delta$ ); /* set  $TA_i$  to  $2\delta$ . */
    /*  $alarm(TA_i)$  will be invoked  $2\delta$  time later unless the timer is reset. */

On  $alarm(TA_i)$ 
  deliver all messages in  $rmsg_i$  in some predefined order;
  clear( $rmsg_i$ );
  set-timer( $TB_i, \delta$ );

```

Fig. 1. The ss-ABcast protocol

**Lemma 1.** *Figure 1 presents a ss-ABcast protocol and the length of each round is at most  $5\delta$ . The ss-ABcast protocol satisfies the following properties: (E1) Validity: If a site invokes  $ABcast(m)$ , then every site eventually invokes  $Deliver(m)$ . (E2) 1-round-stabilizing Integrity: At the second round or later, the Integrity is satisfied: for any message  $m$ , every site invokes  $Deliver(m)$  at most once, and only if some site invoked  $ABcast(m)$ . For the first round, every site invokes  $Deliver(m)$  at most once, if some site invoked  $ABcast(m)$ ; however, some messages may be delivered even if no site broadcasts such messages, henceforth called spurious messages. (E3) 1-round-stabilizing Total Order: At the second round or later, the Total Order is satisfied: if a site invokes  $Deliver(m)$  before  $Deliver(m')$ , then any other site also invokes  $Deliver(m)$  before  $Deliver(m')$ . For the first round, the Total Order is satisfied for the messages that are actually broadcast by sites.*

*Proof Sketch.* (E1) Validity: For contradiction, assume that  $ABcast(m)$  is invoked at a site  $i$  but  $m$  is never delivered at a site  $j$ . We consider two cases. (Case 1)  $m$  is sent but not be delivered: When  $j$  receives  $m$ ,  $j$  sets timer  $TA_j$ . Since  $j$  never delivers  $m$ ,  $TA_j$  never expires after  $j$  receives  $m$ . This implies that  $j$  has no quiet period of length of  $2\delta$  or more after  $j$  receives  $m$ . Let  $\mathcal{M}$  be a set of broadcast messages that are not delivered at  $j$ ,  $\mathcal{M}_1 (\subseteq \mathcal{M})$  be a set of messages (including  $m$ ) that is sent before or at receipt of  $m$  and  $\mathcal{M}_2 = \mathcal{M} - \mathcal{M}_1$ . Let  $t$  be the latest time when a message in  $\mathcal{M}_1$  is sent. No process receives a message

in  $\mathcal{M}_1$  in period  $[t + \delta, \infty]$ . The first sending of a message in  $\mathcal{M}_2$  occurs when  $3\delta$  time passes at a process after the last receipt of a message in  $\mathcal{M}_1$ . Thus, no message in  $\mathcal{M}_2$  is sent in  $[0, t + 3\delta]$ . Therefore, no message is received at any process in  $[t + \delta, t + 3\delta]$  and this contradicts that  $j$  has no quiet period of length of  $2\delta$  or more. (Case 2)  $m$  is not sent (sending of  $m$  is postponed forever): This implies that  $TA_i$  never expires after the invocation of the **ABcast**( $m$ ). This may happen if  $p$  has no quiet period of length of  $2\delta$  or more after the invocation of the **ABcast**. We can show a contradiction by a similar discussion to that of Case 1. It also follows from the above argument that any message  $m$  is delivered within  $5\delta$  from its sending from a site (not from invocation of **ABcast**( $m$ )). Therefore, the length of each round is at most  $5\delta$ . (E2) 1-round-stabilizing Integrity: Spurious messages can be delivered if they are in-transit or stored in *rmsg* at the initial configuration. The in-transit spurious messages are received before  $\delta$  from the beginning of the execution. Thus all spurious messages are delivered at the first round. It is easy to see that the Integrity is guaranteed at the second round or later. (E3) 1-round-stabilizing Total Order: It is sufficient to show that, for any messages  $m_1$  and  $m_2$  actually broadcast, any site delivers  $m_1$  and  $m_2$  in a round if a site delivers them in a round. Let  $i$  be the sender (the originator) of  $m_1$ . From a similar discussion to that of (E1), we can show that  $i$  sends  $m_1$  before or at receiving  $m_2$ . Similarly, the originator of  $m_2$  sends  $m_2$  before or at receiving  $m_1$ . It follows that the times when  $m_1$  and  $m_2$  are sent respectively differ at most  $\delta$ . This implies that these messages are delivered in the same round at any process.  $\square$

#### 4.4 Tree Operation Phase

All that is required for an algorithm to be (self-) stabilizing is eventual convergence to legitimate behavior, however even during the period of convergence to legitimacy, some of the semantics of update operations should be guaranteed (C4): For instance, after an **R\_insert**( $x$ ) operation that responds *ok*, the item  $x$  should be contained in all replicas.

Consider an **R\_insert**( $x$ ) operation applied to a replicated 2-3 tree, which is implemented by **insert**( $x$ ) operations on all replicas. It could be that the **insert**( $x$ ) operation responds *ok* at one site, but it responds *full* at another site, since these sites may start with different initial configurations. If all sites begin with empty trees and all updates are replicated in the same order everywhere, then all 2-3 trees will have the same internal representation of their contents, and the situation of conflicting **insert**( $x$ ) responses will not occur. But if we consider arbitrary initial states, then all replicas could have the same contents, but with different internal representations (to see this, the reader can experiment with different orders of insertion of the same set of items into a 2-3 tree). Thus, it is possible that an **insert**( $x$ ) operation responds *ok* at one site while it responds *full* at another.

To enforce uniformity of **insert** responses, one could maintain a count of items in each replica. Any **insert** operation would respond *full* if the current

count exceeds threshold  $K$ , so that despite the ambiguity of 2-3 tree representation, all  $\text{insert}(x)$  operations would have the same local response. Introducing and relying upon such a counter introduces difficulties because any counter is subject to corruption by a transient fault. Since our goal is to preserve the semantics of any *ok* response during convergence, we reject the idea of a count in favor of another technique.

In our implementation of Fig. 2, an  $\text{R.insert}(x)$  operation has two rounds, a *propose round* and a *response round*. When an  $\text{R.insert}(x)$  operation occurs at site  $i$ , then at each replica  $S_j$  ( $1 \leq j \leq t$ ), the operation  $\text{insert}_j(x)$  is applied, which responds either *ok* or *full* depending on the state of  $S_j$ . In the response round, the response of  $\text{insert}_j(x)$  of every site  $j$  is broadcast to all sites. When a site  $j$  receives the responses from all sites, then it judges whether the  $\text{R.insert}(x)$  should be committed or aborted: if all sites respond *ok*, then the  $\text{insert}_j(x)$  is committed and, moreover, site  $i$  returns *ok* to the outside world (as the response to  $\text{R.insert}(x)$ ); in other cases, the  $\text{insert}_j(x)$  operation is aborted (i.e.,  $\text{delete}_j(x)$  is executed to cancel  $\text{insert}_j(x)$ ) and, moreover, site  $i$  returns *full* to the outside world. Notice that all sites make the same decision, committed or aborted, concerning the  $\text{R.insert}$  operation.

$\text{R.delete}$  and  $\text{R.insert}$  operations require a linearized implementation of updates on the replicas to ensure that tree contents are everywhere the same. Our implementation relies on the  $\text{ss-ABcast}$  protocol in the previous subsection: the propose round and the response round respectively corresponds to one round of the  $\text{ss-ABcast}$  protocol. In the propose round, at most one operation is invoked at each site, and the  $\text{ss-ABcast}$  protocol ensures that the operations are applied in the same order at all sites. However some sites may also apply some operations other than actually invoked ones because of the spurious messages the  $\text{ss-ABcast}$  protocol delivers (but only in the first  $\text{ss-ABcast}$  round). We guarantee availability (C4) of  $\text{R.insert}$  operations by applying all  $\text{delete}_i$  operations before any of  $\text{insert}_i$  operations at each site  $i$ : this prevents spurious  $\text{delete}_i$  operations from deleting the items inserted by  $\text{R.insert}$  operations. In the response round, each process broadcasts a response vector containing all responses of the update operations applied at the previous round. It is not necessary to deliver the response vectors in the same order at all sites, but, for simplicity, our implementation uses the  $\text{ss-ABcast}$  protocol to broadcast the response vectors.

Details of the protocol are presented in Figure 2, and we discuss the main features of the logic here. The protocol deals with only update operations  $\text{R.insert}$  and  $\text{R.delete}$  because  $\text{R.find}$  is an entirely local operation and its implementation is straightforward. In the protocol,  $\text{ABcast}^*$  is used instead of  $\text{ABcast}$  to broadcast operations invoked from the outside world. The difference between the two operations is that  $\text{ABcast}^*$  postpones the broadcast to the propose round of the next tree operation phase if either timer  $TA$  or  $TB$  (in the  $\text{ss-ABcast}$  protocol) is running; recall that  $\text{ABcast}$  simply postpones the broadcast to the next round. This modification is necessary because  $\text{R.insert}$  and  $\text{R.delete}$  may occur at any time but should be dealt with in a tree operation phase.

```

local variables of site  $i$ 
   $op_i$ :array [1.. $t$ ] of operation; /*  $op_i[j]$  is the operation originating at site  $j$  */
   $rsp_i$ :array [1.. $t$ ] of response; /*  $rsp_i[j]$  is the response to  $op_i[j]$  at site  $i$  */
   $rcv\_rsp_i$ :array [1.. $t$ , 1.. $t$ ] of response;
    /*  $rcv\_rsp_i[j, k]$  is the  $rsp_j[k]$  received from site  $j$  */

On receipt of operation  $Op_i$  from the outside world
  /* propose round begins */
  ABcast*( $Op_i$ );
  repeat /* execute local operations in the order delivered */
    wait until Deliver( $op$ );
    (R.delete operations are delivered before any R.insert operation)
    if  $op = \text{R.delete}(x)$  originating at  $j$ 
       $op_i[j] := op$ ;  $\text{delete}_i(x)$ ;  $rsp_i[j] := ok$ ;
    if  $op = \text{R.insert}(x)$  originating at  $j$ 
       $op_i[j] := op$ ;  $\text{insert}_i(x)$ ;  $rsp_i[j] := \text{response to } \text{insert}_i(x)$ ;
  until all messages are delivered;
  /* response round begins */
  ABcast( $rsp_i[1..t]$ );
  repeat /* store response vectors */
    wait until Deliver( $rsp[1..t]$ );
    if  $rsp[1..t]$  is received from  $j$ 
       $rcv\_rsp_i[j, 1..t] := rsp[1..t]$ ;
  until all messages are delivered;
  for each  $k \in [1..t]$  /* determine whether R.insert is committed or aborted */
    if  $op_i[k] = \text{R.insert}(x)$ 
      if  $rcv\_rsp_i[j, k] = \text{full}$  for some  $j$ 
         $\text{delete}_i(x)$ ; /* cancel  $\text{insert}_i(x)$  */
  if  $Op_i = \text{R.insert}(x)$  and  $rcv\_rsp_i[j, i] = \text{full}$  for some  $j$ 
    respond(full)
  else respond(ok)

```

Fig. 2. A protocol for tree operation phase

#### 4.5 Convergence Phase

An illegitimate state for the replicated tree requires repair operations: an arbitrary sequence of **R.insert** and **R.delete** operations need not force all replicas to have identical contents if they initially differ. Therefore, in the convergence phase, additional operations are added. The basic idea is that all replicas participate in a coordinated enumeration of their contents. Suppose this coordinated enumeration begins with the smallest item in each replica: each site broadcasts the smallest item of its replica. If all sites have the same smallest item, then the coordinated enumeration will continue with the second smallest item. This will continue until all sites enumerate the largest tree item, where the enumeration will start again with the smallest tree item. Since at most  $t$  items were newly inserted in each tree operation phase,  $t + 1$  or more items should be enumerated

in each convergence phase to complete the enumeration in time depending on the maximum number of items accessible at a site in the initial configuration. Thus, in our protocol, each site first broadcasts the  $s$  ( $\geq t + 1$ ) smallest items in a convergence phase, and broadcasts the next  $s$  ( $\geq t + 1$ ) smallest items in the next convergence phase, and so on.

As stated in the above, the enumeration goes back to the smallest item when it reaches the maximum one. In the following, however, for simplicity of description, we ignore the situation where such wrapped enumeration occurs.

The only interesting case in the enumeration occurs when sites report a difference for some item. There are two possibilities for such an apparent difference. Either there is a real difference between replicas with respect to the item or sites are somehow uncoordinated in their enumerations. This possibility of an uncoordinated enumeration can arise due to the arbitrary initial configuration. To detect the uncoordinated enumeration, each site broadcasts an item  $z$  called a *starting item* and the  $s$  smallest items larger than or equal to  $z$ . If sites broadcast different starting items, then uncoordinated enumeration is detected and the enumeration starts from the smallest item in each replica by setting  $z$  to  $-\infty$ . If all sites broadcast the same starting items, then coordination in enumeration is guaranteed and differences of the broadcast items imply differences among the contents of replicas. In this case, appropriate insert or delete operations are applied to each replica in order to make correction of the replicas with respect to the items.

It is useful here to depart from the description of our construction and consider various possibilities for correcting the replicas. Two extreme designs are *intersection* and *union*. The intersection approach is to delete item  $x$  from all sites if  $x$  is not present in all sites. The union approach is to insert item  $x$  in all sites if it is present in any of them. But these two approaches are vulnerable to a transient fault: if an item is lost at only a single replica because of a transient fault, the item will be removed from all replicas in the intersection approach. Similarly, the union approach is vulnerable to erroneous insertion at only a single replica. An intermediate and more desirable approach is to use majority vote: delete  $p$  from all replicas unless a majority contain  $p$ , in which case  $p$  should be inserted in all sites. For either the union or majority approach, the remedy of insertion can yet fail: the result of an `insert( $p$ )` operation could be *full*. (the sequential 2-3 tree implementation [10] can respond *full* even when the number of items is less than the threshold capacity during the period of convergence to a legitimate state.) Because of the possibility of encountering *full* responses, we deploy the same two-round idea from the `R.insert` implementation, so sites apply `delete( $p$ )` if  $p$  cannot be inserted in all replicas.

Details of the protocol are shown in Fig. 3. The protocol adopts the majority approach for correcting the replicas, but it can be easily modified so that it should correct the replicas according to some other approach. In the *propose round*, each site finds and broadcasts (using the `ss-ABcast` protocol) the  $s$  ( $\geq t + 1$ ) smallest items larger than or equal to the starting item  $z$ . Note that finding the  $s$  smallest items can be easily implemented by traversing the replica in the



```

local variables of site  $i$ 
   $key_i$ :array [1.. $t$ , 0.. $s$ ] of key;
    /*  $key_i[j, 0]$  is the starting item site  $j$  broadcasts */
    /*  $key_i[j, 1..s]$  is the  $s$  keys site  $j$  broadcasts as
       the  $s$  smallest keys no smaller than  $key[j, 0]$  */
   $start_i$ :key; /* the starting item */
   $ins\_key_i$ :array [1.. $2s$ ] of key; /* the inserted keys */
   $rcv\_rsp_i$ :array [1.. $t$ , 1.. $2s$ ] of response;
    /*  $rcv\_rsp_i[j, m]$  is response to  $insert(ins\_key[m])$  at site  $j$  */

begin
  /* propose round begins */
   $key_i[i, 0] := start_i$ ;
  for each  $m$  ( $1 \leq m \leq s$ )
     $key_i[i, m] :=$ 
      the  $m^{th}$  smallest key no smaller than  $key_i[i, 0]$  in the replica of  $i$ ;
  ABcast( $key_i[i, 0..s]$ );
  repeat /* store keys broadcast by sites */
    wait until Deliver( $key[0..s]$ );
    if  $key[0..s]$  is received from  $j$ 
       $key_i[j, 0..s] := key[0..s]$ ;
  until all messages are delivered;
  if  $key_i[j, 0] \neq key_i[k, 0]$  for some  $j$  and  $k$ 
    /* uncoordinated enumeration is detected */
     $start_i := -\infty$ 
    /* reset the starting item so that enumeration should
       start from the smallest item */
    exit /* skip the correction phase */
  else /* coordinated enumeration is guaranteed */
    apply rules (R1) and (R2) for every item in  $key_i[1..t, 1..s]$ ;
    store the keys satisfying the condition of (R1) in  $ins\_key_i[1..2s]$ ;
    store the corresponding responses in  $rcv\_rsp_i[i, 1..2s]$ ;
     $start_i :=$  the keys chosen by rule (R3);
  /* response round begins */
  ABcast( $rcv\_rsp_i[i, 1..2s]$ );
  repeat /* store the responses broadcast by sites */
    wait until Deliver( $rsp[1..2s]$ );
    if  $rsp[1..2s]$  is received from  $j$ 
       $rcv\_rsp_i[j, 1..2s] := rsp[1..2s]$ ;
  until all messages are delivered;
  for each  $k$  ( $1 \leq k \leq 2s$ )
    if  $rcv\_rsp_i[j, k] = full$  for some  $j$ 
      delete $_i[ins\_key_i[k]]$ ; /* cancel insert $_i$  operation */

```

**Fig. 3.** A protocol for the convergence phase

depth-first fashion from  $z$ . Unless the uncoordinated enumeration is detected, each site  $i$  tries to apply the following two rules to each of the delivered items. In

the rules,  $x$  denotes the item under checking and  $I_j$  ( $1 \leq j \leq t$ ) denotes the set of the  $s$  items site  $i$  received from site  $j$ : **(R1)** If a majority of  $I_j$ 's each contain  $x$ , then site  $i$  executes  $\text{insert}_i(x)$  if  $x$  is not in the  $i$ 's replica. **(R2)** If a majority of  $I_j$ 's each do not contain  $x$  but contain some item larger than  $x$ , then site  $i$  executes  $\text{delete}_i(x)$  if  $x$  is in the  $i$ 's replica. (Notice that the condition of this rule guarantees that only a minority contains  $x$ .) After applying the above rules to all delivered items, the new starting item  $z'$  is determined by the following rule. This rule guarantees that, for any item  $x$  in a replica such that  $z \leq x < z'$  (where  $z$  is the old starting item),  $x$  is contained in all replicas. **(R3)** If every broadcast item satisfies either conditions of (R1) or (R2), then the imaginary key  $x^+$  is chosen as  $z'$  such that  $x^+ > x$  for the largest broadcast item  $x$  and  $x^+ < x'$  for any item  $x' > x$ . Otherwise, the smallest broadcast item that does not satisfy either conditions of (R1) or (R2) is chosen as the new starting item. In the following *response round*, the responses to the  $\text{insert}$  operations executed locally at the site are broadcast (using the  $\text{ss-ABcast}$  protocol), and the  $\text{insert}$  operations are committed or aborted in the same way as the tree operation phase. Notice that the number of the responses each site broadcasts at a response round is less than  $2s$ , because only the responses to  $\text{insert}$  operations are broadcast.

Now we give some intuitive estimation of the number of the  $\text{ss-ABcast}$  rounds for the convergence. Regard that each item is dirty at the initial configuration and becomes clean when it is removed or is guaranteed to be contained in all replicas during the convergence phase. Assume that coordination of enumeration is established. First, consider that, in each convergence phase, each site  $i$  proposes one dirty item  $x_i$  from its replica and applies (R1) and (R2) to the proposed items. If the proposed items have a key as its majority, the rule (R1) makes the majority (at least  $t/2$  items) clean. Otherwise let  $z$  be the new starting point chosen by the rule (R3). Notice that there exist at least  $n/2$  items  $x$  such that  $x \leq z$  among the proposed items. We can see that these items become clean in this or the next convergence phase. In our protocol, each site proposes  $s$  ( $\geq t+1$ ) items in each convergence phase. By amortized estimation, we can regard that each site proposes at least  $s - t$  ( $\geq 1$ ) dirty items in each convergence phase. From the above discussion, two consecutive convergence phases make at least  $t/2$  items clean and, thus, we can show the following theorem.

**Theorem 1.** *The protocol constructed from the protocols of Fig. 2 and Fig. 3 is an implementation of a stabilizing search tree. It reaches a configuration such that all replica have identical contents within  $O(d)$  rounds of the  $\text{ss-ABcast}$  protocol where  $d$  is the maximum number of items accessible at a site in the initial configuration.*

## 5 Conclusion

This paper proposed a construction for available and stabilizing replicated search trees in a message-passing, synchronous distributed system. One of the main contributions of this work is to introduce stabilization and availability into replicated

objects. We have addressed two aspects of availability: `R.find` operations are executed locally, and all operations are allowed, with some reliability guarantees, at all points in an execution. Another contribution of this paper is to present a stabilizing atomic broadcast protocol, a general and powerful tool for designing several services in a stabilizing fashion. Our construction is based on the stabilizing atomic broadcast and checking-and-correction of replicas. This gives a framework for constructing available and stabilizing replicated objects and is expected to be applied for implementing other replicated objects.

## References

1. Y Afek and S Dolev. Local stabilizer. In *Proceedings of the 5th Israeli Symposium on Theory of Computing and Systems*, pages 74–84, 1997.
2. Y Afek, DS Greenberg, M Merritt, and G Taubenfeld. Computing with faulty shared objects. *Journal of the Association of the Computing Machinery*, 42:1231–1274, 1995.
3. A Bui, AK Datta, F Petit, and V Villain. State-optimal snap-stabilizing PIF in tree networks. In *Proceedings of the Third Workshop on Self-Stabilizing Systems (published in association with ICDCS99 The 19th IEEE International Conference on Distributed Computing Systems)*, pages 78–85. IEEE Computer Society, 1999.
4. S Dolev and T Herman. Superstabilizing protocols for dynamic distributed systems. *Chicago Journal of Theoretical Computer Science*, 3(4), 1997.
5. M Fischer, S Moran, S Rudich, and G Taubenfeld. The wakeup problem. In *STOC90 Proceedings of the 22th Annual ACM Symposium on Theory of Computing*, pages 106–116, 1990.
6. S Ghosh, A Gupta, T Herman, and SV Pemmaraju. Fault-containing self-stabilizing algorithms. In *PODC96 Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 45–54, 1996.
7. V Hadzilacos and S Toueg. Fault-tolerant broadcasts and related problems. In S. Mullender (ed.) *"Distributed Systems (2nd ed.)"*, Addison-Wesley, 1993.
8. T Herman. Superstabilizing mutual exclusion. *Distributed Computing*, 13(1):1–17, 2000.
9. T Herman and T Masuzawa. Available stabilizing heaps. *Information Processing Letters* 77, pp. 115–121, 2001.
10. T Herman and T Masuzawa. A stabilizing search tree with availability properties. *Proceedings of the Fifth International Symposium on Autonomous Decentralized Systems (ISADS'01)*, pp. 398–405, March 2001.
11. JH Hoepman, M Papatriantafilou, and P Tsigas. Self-stabilization of wait-free shared memory objects. In *WDAG95 Distributed Algorithms 9th International Workshop Proceedings, Springer-Verlag LNCS:972*, pages 273–287, 1995.
12. S Kutten and B Patt-Shamir. Stabilizing time-adaptive protocols. *Theoretical Computer Science*, 220:93–111, 1999.
13. N Lynch. *Distributed Algorithms*. Morgan Kaufmann, 1996.
14. E Ueda, Y Katayama, T Masuzawa, and H Fujiwara. A latency-optimal super-stabilizing mutual exclusion protocol. In *Proceedings of the Third Workshop on Self-Stabilizing Systems*, pages 110–124. Carleton University Press, 1997.

# Adding Networks<sup>\*</sup>

Panagiota Fatourou<sup>1</sup> and Maurice Herlihy<sup>2</sup>

<sup>1</sup> Department of Computer Science, University of Toronto,  
10 King's College Road, Toronto, Canada M5S 3G4  
`faturu@cs.toronto.edu`

<sup>2</sup> Department of Computer Science, Brown University,  
Box 1910, Providence, RI 02912-1910  
`mph@cs.brown.edu`

**Abstract.** An *adding network* is a distributed data structure that supports a concurrent, lock-free, low-contention implementation of a *fetch&add* counter; a counting network is an instance of an adding network that supports only *fetch&increment*.

We present a lower bound showing that adding networks have inherently high latency. Any adding network powerful enough to support addition by at least two values  $a$  and  $b$ , where  $|a| > |b| > 0$ , has sequential executions in which each token traverses  $\Omega(n/c)$  switching elements, where  $n$  is the number of concurrent processes, and  $c$  is a quantity we call *one-shot contention*; for a large class of switching networks and for conventional counting networks the one-shot contention is constant. On the contrary, counting networks have  $O(\log n)$  latency [4,7].

This bound is tight. We present the first concurrent, lock-free, low-contention networked data structure that supports arbitrary fetch&add operations.

## 1 Introduction

### Motivation-Overview

A *fetch&increment* variable provides an operation that atomically adds one to its value and returns its prior value. Applications of *fetch&increment* counters include shared pools and stacks, load balancing, and software barriers.

A *counting network* [3] is a class of distributed data structures used to construct concurrent, low-contention implementations of *fetch&increment* counters. A limitation of the original counting network constructions is that the resulting shared counters can be incremented, but not decremented. More recently, Shavit and Touitou [11] showed how to extend certain counting network constructions to support decrement operations, and Aiello and others [2] extended this technique to arbitrary counting network constructions.

---

<sup>\*</sup> This work has been accepted for publication as a brief announcement in the *20th Annual ACM Symposium on Principles of Distributed Computing*, Newport, Rhode Island, August 2001. Part of the work of the first author was performed while affiliating with the Max-Planck Institut für Informatik, Saarbrücken, Germany, and while visiting the Department of Computer Science, Brown University, Providence, USA.

In this paper we consider the following natural generalization of these recent results: can we construct network data structures that support lock-free, highly-concurrent, low-contention *fetch&add* operations? (A *fetch&add* atomically adds an *arbitrary* value to a shared variable, and returns the variable's prior value.)

We address these problems in the context of concurrent *switching networks*, a generalization of the balancing networks used to construct counting networks. As discussed in more detail below, a switching network is a directed graph, where edges are called *wires* and nodes are called *switches*. Each of the  $n$  processes shepherds a *token* through the network. Switches and tokens are allowed to have internal states. A token arrives at a switch via an input wire. In one atomic step, the switch absorbs the token, changes its state and possibly the token's state, and emits the token on an output wire.

Let  $S$  be any non-empty set of integer values. An  $S$ -*adding network* is a switching network that implements the set of operations **fetch&add**( $\cdot, s$ ), for  $s$  an element of  $S$ . As a special case, an  $(a, b)$ -adding network supports two operations: **fetch&add**( $\cdot, a$ ) and **fetch&add**( $\cdot, b$ ). A process executes a **fetch&add**( $\cdot, a$ ) operation by shepherding a token of *weight*  $a$  through the network.

Our results encompass both bad news and good news. First the bad news. We define the network's *one-shot contention* to be the largest number of tokens that can meet at a single switch in any execution in which exactly  $n$  tokens enter the network on distinct wires. For counting networks, and for the (first) switching network presented in Section 4, this quantity is constant. We show that for any  $(a, b)$ -adding network, where  $|a| > |b| > 0$ , there exist  $n$ -process sequential executions where each process traverses  $\Omega(n/c)$  switches, where  $c$  is the network's one-shot contention. This result implies that any lock-free low-contention adding network must have high worst-case latency, even in the absence of concurrency. As an aside, we note that there are two interesting cases not subject to our lower bound: a low-latency  $(a, -a)$ -adding network is given by the antitoken construction, and an  $(a, 0)$ -adding network is just a regular counting network augmented by a pure read operation.

Now for the good news. We introduce a novel construction for a lock-free, low-contention *fetch&add* switching network, called LADDER, in which processes take  $O(n)$  steps on average. Tokens carry mutable values, and switching elements are balancers augmented by atomic read-write variables. The construction is lock-free, but not wait-free (meaning that individual tokens can be overtaken arbitrarily often, but that some tokens will always emerge from the network in a finite number of steps). LADDER is the first concurrent, lock-free, low-contention networked data structure that supports arbitrary *fetch&add* operations.

An ideal *fetch&add* switching network (like an ideal counting network defined in [6]) is (1) lock-free, with (2) low contention, and (3) low latency. Although this paper shows that no switching network can have all three properties, any two are possible. 1 a single switch is lock-free with low latency, but has high contention, a combining network [5] has low contention and  $O(\log n)$  latency but requires

<sup>1</sup> NASA's motto "faster, cheaper, better" has been satirized as "faster, cheaper, better: pick any two".

tokens to wait for one another, and the construction presented here is lock-free with low contention, but has  $O(n)$  latency.

## Related Work

Counting networks were first introduced by Aspnes *et. al* [3]. A flurry of research on counting networks followed (see e.g., [1, 2, 4, 6, 7, 9, 11]). Counting networks are limited to support only *fetch&increment* and *fetch&decrement* operations. Our work is the first to study whether lock-free network data structures can support even more complex operations. We generalize traditional counting networks by introducing switching networks, which employ more powerful switches and tokens. Switches can be shared objects characterized by arbitrary internal states. Moreover, each token is allowed to have a state by maintaining its own variables; tokens can exchange information with the switches they traverse.

Surprisingly, it turns out that supporting even the slightly more complex operation of *fetch&add*, where adding is by only two different integers  $a, b$  such that  $|a| > |b| > 0$ , is as difficult as ensuring linearizability [6]. In [6] the authors prove that there exists no ideal linearizable counting network. In a corresponding way, our lower bound implies that even the most powerful switching networks cannot guarantee efficient support of this relatively simple *fetch&add* operation.

The LADDER switching network has the same topology as the linearizable SKEW presented by Herlihy and others [6], but the behavior of the LADDER network is significantly different. In this network, tokens accumulate state as they traverse the network, and they use that state to determine how they interact with switches. The resulting network is substantially more powerful, and requires a substantially different analysis.

## Organization

This paper is organized as follows. Section 2 introduces switching networks. Our lower bound is presented in Section 3, while the **Ladder** network is introduced in Section 4.

## 2 Switching Networks

A *switching network*, like a counting network [3], is a directed graph whose nodes are simple computing elements called *switches*, and whose edges are called *wires*. A wire directed from switch  $b$  to switch  $b'$  is an *output wire* for  $b$  and an *input wire* for  $b'$ . A  $(w_{in}, w_{out})$ -switching network has  $w_{in}$  input wires and  $w_{out}$  output wires. A *switch* is a shared data object characterized by an internal state, its set of  $f_{in}$  *input wires*, labeled  $0, \dots, f_{in} - 1$ , and its set of  $f_{out}$  *output wires*, labeled  $0, \dots, f_{out} - 1$ . The values  $f_{in}$  and  $f_{out}$  are called the switch's *fan-in* and *fan-out* respectively.

There are  $n$  processes that move (shepherd) *tokens* through the network. Each process enters its token on one of the network's  $w_{in}$  input wires. After the

token has traversed a sequence of switches, it leaves the network on one of its  $w_{out}$  output wires. A process shepherds only one token at a time, but it can start shepherding a new token as soon as its previous token has emerged from the network. Processes work asynchronously, but they do not fail. In contrast to counting networks, associated to each token is a set of variables (that is, each token has a mutable state), which can change as it traverses the network.

A switch acts as a router for tokens. When a token arrives on a switch's input wire, the following events can occur atomically: (1) the switch removes the token from the input wire, (2) the switch changes state, (3) the token changes state, and (4) the switch places the token on an output wire. The wires are one-way communication channels and allow reordering. Communication is asynchronous but reliable (meaning a token does not wait on a wire forever). For each  $(f_{in}, f_{out})$ -switch, we denote by  $x_i$ ,  $0 \leq i \leq f_{in} - 1$ , the number of tokens that have entered on input wire  $i$ , and similarly we denote by  $y_j$ ,  $0 \leq j \leq f_{out} - 1$ , the number of tokens that have exited on output wire  $j$ .

As an example, a  $(k, \ell)$ -balancer is a switch with fan-in  $k$  and fan-out  $\ell$ . The  $i$ -th input token is routed to output wire  $i \bmod \ell$ . Counting networks are constructed from balancers and from simple one-input one-output counting switches.

It is convenient to characterize a switch's internal state as a collection of variables, possibly with initial values. The state of a switch is given by its internal state and the collection of tokens on its input and output wires. Each token's state is also characterized by a set of variables. Notice that a token's state is part of the state of the process owning it. A process may change the state of its token while moving it through a switch. A switching network's state is just the collection of the states of its switches.

A switch is *quiescent* if the number of tokens that arrived on its input wires equals the number that have exited on its output wires:  $\sum_{i=0}^{f_{in}-1} x_i = \sum_{j=0}^{f_{out}-1} y_j$ . The *safety property* of a switch states that in any state,  $\sum_{i=0}^{f_{in}-1} x_i \geq \sum_{j=0}^{f_{out}-1} y_j$ ; that is, a switch never creates tokens spontaneously. The *liveness property* states that given any finite number of input tokens to the switch, it is guaranteed that it will eventually reach a quiescent state. A switching network is *quiescent* if all its switches are quiescent.

We denote by  $\pi = \langle t, b \rangle$  the *state transition* in which the token  $t$  is moved from an input wire to an output wire of a switch  $b$ . If a token  $t$  is on one of the input wires of a switch  $b$  at some network state  $s$ , we say that  $t$  is *in front of*  $b$  at state  $s$  or that the transition  $\langle t, b \rangle$  is *enabled* at state  $s$ . An *execution fragment*  $\alpha$  of the network is either a finite sequence  $s_0, \pi_1, s_1, \dots, \pi_n, s_n$  or an infinite sequence  $s_0, \pi_1, s_1, \dots$  of alternating network states and transitions such that for each  $\langle s_i, \pi_{i+1}, s_{i+1} \rangle$ , the transition  $\pi_{i+1}$  is enabled at state  $s_i$  and carries the network to state  $s_{i+1}$ . If  $\pi_{i+1} = \langle t, b \rangle$  we say that token  $t$  *takes a step* at state  $s_i$  (or that  $t$  *traverses*  $b$  at state  $s_i$ ). An execution fragment beginning with an initial state is called an *execution*. If  $\alpha$  is a finite execution fragment of the network and  $\alpha'$  is any execution fragment that begins with the last state of  $\alpha$ , then we write  $\alpha \cdot \alpha'$  to represent the sequence obtained by concatenating  $\alpha$  and  $\alpha'$  and eliminating the duplicate occurrence of the last state of  $\alpha$ .

For any token  $t$ , a  $t$ -solo execution fragment is an execution fragment in all transitions of which token  $t$  only takes steps. A  $t$ -complete execution fragment is an execution fragment at the final state of which token  $t$  has exited the network. A finite execution is *complete* if it results in a quiescent state. An execution is *sequential* if for any two transitions  $\pi = \langle t, b \rangle$  and  $\pi' = \langle t, b' \rangle$ , all transitions between them also involve token  $t$ ; that is, tokens traverse the network one completely after the other.

A switch  $b$  has the  $l$ -balancing property if, whenever  $l$  tokens reach each input wire of  $b$  then exactly  $l$  tokens exit on each of its output wires. We say that a switching network is an  $l$ -balancing network if all its switches preserve the  $l$ -balancing property. It can be proved [6] that in any execution  $\alpha$  of an  $l$ -balancing network  $\mathcal{N}$ , in which no more than  $l$  tokens enter on any input wire of the network, there are never more than  $l$  tokens on any wire of the network.

The *latency* of a switching network is the maximum number of switches traversed by any single token in any execution. The *contention of an execution* is the maximum number of tokens that are on the input wires of any particular switch at any point during the execution. The *contention* of a switching network is the maximum contention of any of its executions. In a *one-shot* execution, only  $n$  tokens (one per process) traverse the network. The *one-shot contention* of a switching network, denoted  $c$ , is the maximum contention over all its one-shot executions in which the  $n$  tokens are uniformly distributed on the input wires. For counting networks with  $\Omega(n)$  input wires, and for the switching network presented in Section 4,  $c$  is constant.

For any integer set  $S$ , an  $S$ -adding network  $\mathcal{A}$  is a switching network that supports the operation  $\text{fetch\&add}(\cdot, v)$  only for values  $v \in S$ . More formally, let  $l > 0$  be any integer and consider any complete execution  $\alpha$  which involves  $l$  tokens  $t_1, \dots, t_l$ . Assume that for each  $i$ ,  $1 \leq i \leq l$ ,  $\beta_i$  is the weight of  $t_i$  and  $v_i$  is the value taken by  $t_i$  in  $\alpha$ . The *adding property* for  $\alpha$  states that there exists a permutation  $i_1, \dots, i_l$  of  $1, \dots, l$ , called the *adding order*, such that (1)  $v_{i_1} = 0$ , and (2) for each  $j$ ,  $1 \leq j < l$ ,  $v_{i_{j+1}} = v_{i_j} + \beta_{i_j}$ ; that is, the first token in the order returns the value zero, and each subsequent token returns the sum of the weights of the tokens that precede it. We say that a switching network is an *adding network* if it is a  $\mathbb{Z}$ -adding network, where  $\mathbb{Z}$  is the set of integers.

### 3 Lower Bound

Consider an  $(a, b)$ -adding network such that  $|a| > |b| > 0$ . We may assume without loss of generality that  $a$  and  $b$  have no common factors, since any  $(a, b)$ -adding network can be trivially transformed to an  $(a \cdot k, b \cdot k)$ -adding network, and vice-versa, for any non-zero integer  $k$ . Similarly, we can assume that  $a$  is positive. We show that in *any* sequential execution (involving any number of tokens), tokens of weight  $b$  must traverse at least  $\lceil (n-1)/(c-1) \rceil$  switches, where  $c$  is the one-shot contention of the network. If  $|b| > 1$ , then in any sequential execution, tokens of weight  $a$  must also traverse the same number of switches.



We remark that our lower bound holds for all  $(a, b)$ -adding networks, independently of e.g., the topology of the network (the width or the depth of the network, etc.) and the state of both the switches and the tokens. Moreover, our lower bound holds for *all* sequential executions involving any number of tokens (and not only for one-shot executions).

**Theorem 1.** *Consider an  $(a, b)$ -adding network  $\mathcal{A}$  where  $|a| > |b| > 0$ . Then, in any sequential execution of  $\mathcal{A}$ ,*

1. *each token of weight  $b$  traverses at least  $\lceil (n-1)/(c-1) \rceil$  switches, and*
2. *if  $|b| > 1$  then each token of weight  $a$  traverses at least  $\lceil (n-1)/(c-1) \rceil$  switches.*

*Proof.* We prove something slightly stronger, that the stated lower bound holds for any token that goes through the network alone, independently of whether tokens before it have gone through the network sequentially.

Start with the network in a quiescent state  $s_0$ , denote by  $\alpha_0$  the execution with final state  $s_0$ , and let  $t'_1, \dots, t'_l$  be the tokens involved in  $\alpha_0$ , where  $l \geq 0$  is some integer. Denote by  $\beta_j$ ,  $1 \leq j \leq l$ , the weight of token  $t'_j$  and let  $v = \sum_{j=1}^l \beta_j$ . The adding property implies that  $v$  is the next value to be taken by any token (serially) traversing the network. Let token  $t$  of weight  $x$ ,  $x \in \{a, b\}$ , traverse the network next. Let  $y \in \{a, b\}$ ,  $y \neq x$ , be the other value of  $\{a, b\}$ ; that is, if  $x = a$  then  $y = b$ , and vice versa. Because  $|a| > |b| > 0$ ,  $b \not\equiv 0 \pmod{a}$ . Thus, if  $x = b$ ,  $x \not\equiv 0 \pmod{y}$ . On the other hand, if  $x = a$  it again holds that  $x \not\equiv 0 \pmod{y}$  because by assumption  $|y| > 1$  and  $x, y$  have no common factors.

Denote by  $\mathcal{B}$  the set of switches that  $t$  traverses in a  $t$ -solo,  $t$ -complete execution fragment from  $s_0$ .

Consider  $n-1$  tokens  $t_1, \dots, t_{n-1}$ , all of weight  $y$ . We construct an execution in which each token  $t_i$ ,  $1 \leq i \leq n-1$ , must traverse some switch of  $\mathcal{B}$ . Assume that all  $n$  tokens  $t, t_1, \dots, t_{n-1}$  are uniformly distributed on the input wires.

**Lemma 1.** *For each  $i$ ,  $1 \leq i \leq n-1$ , there exists a  $t_i$ -solo execution fragment with final state  $s_i$  starting from state  $s_{i-1}$  such that  $t_i$  is in front of a switch  $b_i \in \mathcal{B}$  at state  $s_i$ .*

*Proof.* By induction on  $i$ ,  $1 \leq i \leq n-1$ .

#### Basis Case

We claim that in the  $t_1$ -solo,  $t_1$ -complete execution fragment  $\alpha'_1$  starting from state  $s_0$ , token  $t_1$  traverses at least one switch of  $\mathcal{B}$ . Suppose not. Denote by  $s'_1$  the final state of  $\alpha'_1$ . Because  $\mathcal{A}$  is an adding network,  $t_1$  takes the value  $v$  in  $\alpha'_1$ .

Consider now the  $t$ -solo,  $t$ -complete execution fragment  $\alpha''_1$  starting from state  $s'_1$ . Since  $t_1$  does not traverse any switch of  $\mathcal{B}$ , all switches traversed by  $t$  in  $\alpha''_1$  have the same state in  $s_0$  and  $s'_1$ . Therefore, token  $t$  takes the same value in  $\alpha''_1$  as in the  $t$ -solo,  $t$ -complete execution fragment starting from  $s_0$ . It follows that  $t$  takes the value  $v$ .

We have constructed an execution in which both tokens  $t$  and  $t_1$  take the value  $v$ . Since  $|a|, |b| > 0$ , this contradicts the adding property of  $\mathcal{A}$ .

It follows that  $t_1$  traverses at least one switch of  $\mathcal{B}$  in  $\alpha'_1$ . Let  $\alpha_1$  be the shortest prefix of  $\alpha'_1$  such that  $t_1$  is in front of a switch  $b_1 \in \mathcal{B}$  at the final state  $s_1$  of  $\alpha_1$ .

### Induction Hypothesis

Assume inductively that for some  $i$ ,  $1 < i \leq n-1$ , the claim holds for all  $j$ ,  $1 \leq j < i$ ; that is, there exists an execution fragment  $a_j$  with final state  $s_j$  starting from state  $s_{j-1}$  such that token  $t_j$  is in front of a switch  $b_j \in \mathcal{B}$  at state  $s_j$ .

### Induction Step

For the induction step, we prove that in the  $t_i$ -solo,  $t_i$ -complete execution fragment  $\alpha'_i$  starting from state  $s_{i-1}$ , token  $t_i$  traverses at least one switch of  $\mathcal{B}$ . Suppose not. Denote by  $s'_i$  the final state of  $\alpha'_i$ . Since  $t$  has taken no step in execution  $\alpha_1 \cdot \dots \cdot \alpha_{i-1}$ , the adding property of  $\mathcal{A}$  implies that token  $t_i$  takes value  $v_i \equiv v \pmod{y}$ . Consider now the  $t$ -solo,  $t$ -complete execution fragment  $\alpha''_i$  starting from state  $s'_i$ . By construction of the execution  $\alpha_1 \cdot \dots \cdot \alpha_{i-1}$ , tokens  $t_1, \dots, t_{i-1}$  do not traverse any switch of  $\mathcal{B}$  in  $\alpha_1 \cdot \dots \cdot \alpha_{i-1}$ . Therefore, all switches traversed by  $t$  in  $\alpha''_i$  have the same state at  $s_0$  and  $s'_i$ . Thus, token  $t$  takes the value  $v$  in both  $\alpha''_i$  and in the  $t$ -solo,  $t$ -complete execution fragment starting from  $s_0$ .

Because  $\mathcal{A}$  is an adding network, if  $t$  takes the value  $v$ , then  $t_i$  must take value  $v_i \equiv v + x \pmod{y}$ , but we have just constructed an execution where  $t$  takes value  $v$ , and  $t_i$  takes value  $v_i \equiv v \pmod{y}$ , which is a contradiction because  $x \not\equiv 0 \pmod{y}$ . Thus, token  $t_i$  traverses at least one switch of  $\mathcal{B}$  in  $\alpha'_i$ . Let  $\alpha_i$  be the shortest prefix of  $\alpha'_i$  such that  $t_i$  is in front of a switch  $b_i \in \mathcal{B}$  at the final state  $s_i$  of  $\alpha_i$ , to complete the proof of the induction step.

At this point the proof of Lemma [11](#) is complete.

Let  $\alpha = \alpha_1 \cdot \dots \cdot \alpha_{n-1}$ . Clearly, only  $n$  tokens, one per process, are involved in  $\alpha$  and they are uniformly distributed on the input wires, so  $\alpha$  is a one-shot execution. By Lemma [11](#), all tokens  $t_i$ ,  $1 \leq i \leq n-1$  are in front of switches of  $\mathcal{B}$  at the final state of  $\alpha$ . Notice also that all switches in  $\mathcal{B}$  are in the same state at states  $s_0$  and  $s_{n-1}$ . Thus, in the  $t$ -solo,  $t$ -complete execution fragment starting from state  $s_{n-1}$  token  $t$  traverses all switches of  $\mathcal{B}$ . Because  $\mathcal{A}$  has one-shot contention  $c$ , no more than  $c-1$  other tokens can be in front of any switch of  $\mathcal{B}$  in  $\alpha$ . Thus,  $\mathcal{B}$  must contain at least  $\lceil \frac{n-1}{c-1} \rceil$  switches.

Since any  $S$ -adding network, where  $|S| > 2$ , is an  $S'$ -adding network for all  $S' \subseteq S$ , Theorem [11](#) implies that in every sequential execution of the  $S$ -adding network all tokens (except possibly those with maximum weight) traverse  $\Omega(n/c)$  switches.

We remark that the one-shot contention  $c$  of a large class of switching networks, including conventional counting networks, is constant. For example, consider the class of switching networks with  $\Omega(n)$  input wires whose switches produce any permutation of their input tokens on their output wires. A straightforward induction argument shows that each switching network of this class has the 1-balancing property, and thus in a one-shot execution it never has more

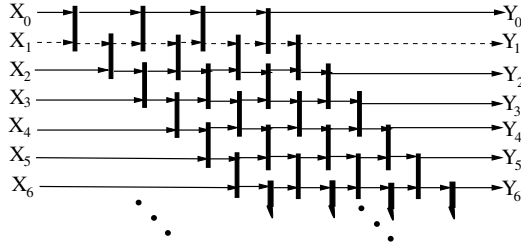


Fig. 1. The **Ladder** switching network of layer depth 4.

than one token on each wire. It follows that the one-shot contention of such a network is bounded by the maximum fan-in of any of its switches. By Theorem 1, all adding networks in this class, have  $\Omega(n)$  latency. Conventional counting networks (and the **LADDER** adding network introduced in Section 4) belong to this class.

## 4 Upper Bound

In this section, we show that the lower bound of Section 3 is essentially tight. We present a low-contention adding network, called **LADDER**, such that in any of its sequential executions tokens traverse  $O(n)$  switches, while in its concurrent executions they traverse an *average* of  $O(n)$  switches. The switching network described here has the same topology as the **SKEW** counting network [6], though its behavior is substantially different. A **Ladder layer** is an unbounded-depth switching network consisting of a sequence of *binary* switches  $b_i$ ,  $i \geq 0$ , that is, switches with  $f_{in} = f_{out} = 2$ . For switch  $b_0$ , both input wires are input wires to the layer, while for each switch  $b_i$ ,  $i > 0$ , the first (or *north*) input wire is an output wire of switch  $b_{i-1}$ , while the second (or *south*) input wire is an input wire of the layer. The north output wire of any switch  $b_i$ ,  $i \geq 0$ , is an output wire of the layer, while the south output wire of  $b_i$  is the north input wire of switch  $b_{i+1}$ .

A **Ladder switching network of layer depth  $d$**  is a switching network constructed by layering  $d$  **Ladder** layers so that the  $i$ -th output wire of the one is the  $i$ -th input wire to the next. Clearly, the **Ladder** switching network has an infinite number of input and output wires. The **LADDER adding network** consists of a counting network followed by a **Ladder** switching network of layer depth  $n$ .

Figure 1 illustrates a **Ladder** switching network of layer depth 4. Switches are represented by fat vertical lines, while wires by horizontal arrows. Wires  $X_0, X_1, \dots$ , are the input wires of the network, while  $Y_0, Y_1, \dots$ , are its output wires. All switches for which one of their input wires is an input wire of the network belong to the first **Ladder** layer. All dashed wires belong to row 1.

Each process moves its token through the counting network first, and uses the result to choose an input wire to the **Ladder** network. The counting network ensures that each input wire is chosen by exactly one token, and each switch is

visited by two tokens. A *fresh* switch is one that has never been visited by a token. Each switch  $s$  has the following state: a bit  $s.toggle$  that assumes values **north** and **south**, initially **north**, and an integer value  $s.weight$ , initially 0. The fields  $s.north$  and  $s.south$  are pointers to the (immutable) switches that are connected to  $s$  through its north and south output wires, respectively.

Each token  $t$  has the following state: the  $t.arg$  field is the original weight of the token. The  $t.weight$  field is originally 0, and it accumulates the sum of the weights of tokens ordered before  $t$ . The  $t.wire$  field records whether the token will enter the next switch on its north or south input wire.

Within **Ladder**, a token proceeds in two *epochs*, a *north* epoch followed by a *south* epoch. Tokens behave differently in different epochs. A token's north epoch starts when the token enters **Ladder**, continues as long as it traverses fresh switches, and ends as soon as it traverses a non-fresh switch. When a north-epoch token visits a fresh switch, the following occurs atomically (1)  $s.toggle$  flips from **north** to **south**, and (2)  $s.weight$  is set to  $t.weight + t.arg$ . Then,  $t$  exits on the switch's north wire.

The first time a token visits a non-fresh switch, it adds that switch's weight to its own, exits on the south wire, and enters its south epoch. Once a token enters its south epoch, it never moves "up" to a lower-numbered row. When a south-epoch token enters a switch on its south wire, it simply exits on the same wire (and same row), independently of the switch's current state. When a south-epoch token enters a switch on its north wire, it does the following. If the switch is fresh, then, as before, it atomically sets the switch's weight to the sum of its weight and argument, flips the toggle bit, and exits on the north wire (same row). If the switch is not fresh, it adds the switch's weight to its own, and exits on the south wire (one row "down"). When the token emerges from **LADDER**, its current weight is its output value.

All tokens other than the one that exits on the first output wire eventually reach a non-fresh switch. When a token  $t$  encounters its first non-fresh switch, then that switch's weight is the sum of all the tokens that will precede  $t$  (so far) in the adding order. Each time the token enters a non-fresh switch on its north wire, it has been "overtaken" by an earlier token, so it moves down one row and adds this other token's weight to its own. Figure 2 shows pseudo-code for the two epochs. For ease of presentation, the pseudocode shows the switch complementing its *toggle* field and updating its *weight* field in one atomic operation. However, a slightly more complicated construction can realize this state change as a simple atomic complement operation on the toggle bit.

Even though **Ladder** has an unbounded number of switches, it can be implemented by a finite network by "folding" the network so that each folded switch simulates an unbounded number of primitive switches. A similar folding construction appears in [6].

**LADDER** is lock-free, but not wait-free. It is possible for a slow token to remain in the network forever if it is overtaken by infinitely many faster tokens.

Proving that **LADDER** is an adding network is a major challenge of our analysis. We point out that although **LADDER** has the same topology as **SKEW** [6],

```

void north_traverse(token t, switch s) {
    if (s.toggle == NORTH) { /* fresh */
        atomically {
            s.toggle = SOUTH;
            s.weight = t.weight + t.arg;
        }
        north_traverse(t, b.north);
    } else { /* not-so-fresh */
        t.weight += s.weight;
        t.wire = NORTH;
        south_traverse(t, b.south);
    }
}

void south_traverse(token t, switch s) {
    if (t.wire == SOUTH) { /* ignore switch */
        t.wire = NORTH; /* toggle wire */
        south_traverse(t, s.south);
    } else {
        t.wire = SOUTH; /* toggle wire */
        if (s.toggle == NORTH) { /* fresh */
            atomically {
                s.toggle = SOUTH;
                s.weight = t.weight + t.arg;
            }
            south_traverse(t, s.north);
        } else { /* overtaken */
            t.weight += s.weight;
            south_traverse(t, s.south);
        }
    }
}

```

**Fig. 2.** Pseudo-Code for LADDER Traversal.

LADDER is substantially more powerful than SKEW; we require a substantially different and more complicated analysis to prove its adding property.

**Theorem 2.** *LADDER is an adding network.*

The performance analysis of LADDER uses similar arguments as the one of SKEW. This follows naturally from the fact that the two networks have the same topology and they both maintain the 1-balancing property (notice that, by knowing just the topology of a switching network, it is not always possible to analyze its performance because the way each token moves in the network may depend on both the state of the token and the state of any switch it traverses).

It can be proved that Ladder can itself be used to play the role of the conventional counting network. From now on, we assume that this is the case, that

is, the LADDER adding network consists only of the Ladder switching network (which it also uses as a traditional counting network).

- Theorem 3.** (a) *In any execution of LADDER, each token traverses an average number of  $2n$  switches;*  
 (b) *In any sequential execution of LADDER, each token traverses exactly  $2n$  switches;*  
 (c) *The contention of LADDER is 2.*

An execution of a switching network is *linearizable* if for any two tokens  $t$  and  $t'$  such that  $t'$  entered the network after  $t$  has exited it, it holds that  $v_t < v_{t'}$ , where  $v_t, v_{t'}$  are the output values of  $t$  and  $t'$ , respectively. For any adding network *there exist* executions which are linearizable (e.g., executions in which all tokens have different weights which are powers of two). For LADDER it holds that *any* of its executions is linearizable. It has been proved [6, Theorem 5.1, Section 5] that any non-blocking linearizable counting network other than the trivial network with only one balancer has infinite number of input wires; that is, if *all* the executions of the network are linearizable, then the network has infinite width. Although an adding network implements a *fetch&increment* operation (and thus it can serve as a counting network), this lower bound does not apply for adding networks because its proof uses the fact that counting networks consist only of balancers and counter objects which is not generally the case for switching networks.

## 5 Discussion

We close with some straightforward generalizations of our results. Consider a family  $\Phi$  of functions from values to values. Let  $\phi$  be an element of  $\Phi$  and  $x$  a variable. The *read-modify-write* operation [8],  $RMW(x, \phi)$ , atomically replaces the value of  $x$  with  $\phi(x)$ , and returns the prior value of  $x$ . Most common synchronization primitives, such as *fetch&add*, *swap*, *test&set*, and *compare&swap*, can be cast as *read-modify-write* operations for suitable choices of  $\phi$ . A *read-modify-write network* is one that supports *read-modify-write* operations. The LADDER network is easily extended to a read-modify-write network for any family of *commutative* functions (for all functions  $\phi, \psi \in \Phi$ , and all values  $v$ ,  $\phi$  and  $\psi$  are *commutative*, if and only if  $\phi(\psi(v)) = \psi(\phi(v))$ ). A map  $\phi$  can *discern* another map  $\psi$  if  $\phi^k(\psi(x)) \neq \phi^\ell(x)$  for some value  $x$  and all natural numbers  $k$  and  $\ell$ . Informally, one can always tell whether  $\psi$  has been applied to a variable, even after repeated successive applications of  $\phi$ . For example, if  $\phi$  is addition by  $a$  and  $\psi$  addition by  $b$ , where  $|a| > |b| > 0$ , then  $\phi$  can discern  $\psi$ . Our lower bound can be generalized to show that if a switching network supports read-modify-write operations for two functions one of which can discern the other, then in any  $n$ -process sequential execution, processes traverse  $\Omega(n/c)$  switches before choosing a value.

Perhaps the most important remaining open question is whether there exist low-contention wait-free adding networks (notice that since switching networks

do not contain cycles, any network with a finite number of switches would be wait-free).

**Acknowledgement.** We would like to thank Faith Fich for many useful comments that improved the presentation of the paper. Our thanks go to the anonymous DISC'01 reviewers for their feedback.

## References

1. Aharonson, E., Attiya, H.: Counting networks with arbitrary fan-out. *Distributed Computing*, **8** (1995) 163–169.
2. Aiello, W., Busch, C., Herlihy, M., Mavronicolas, M., Shavit, N., Touitou, D.: Supporting Increment and Decrement Operations in Balancing Networks. *Proceedings of the 16th International Symposium on Theoretical Aspects of Computer Science*, pp. 393–403, Trier, Germany, March 1999.
3. Aspnes, J., Herlihy, M., Shavit, N.: Counting Networks. *Journal of the ACM*, **41** (1994) 1020–1048.
4. Busch, C., Mavronicolas, M.: An Efficient Counting Network. *Proceedings of the 1st Merged International Parallel Processing Symposium and IEEE Symposium on Parallel and Distributed Processing*, pp. 380–385, Orlando, Florida, May 1998.
5. Goodman, J., Vernon, M., Woest, P.: Efficient synchronization primitives for large-scale cache-coherent multiprocessors. *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 64–75, Boston, Massachusetts, April 1989.
6. Herlihy, M., Shavit, N., Waarts, O.: Linearizable Counting Networks. *Distributed Computing*, **9** (1996) 193–203.
7. Klugerman, M., Plaxton, C.: Small-Depth Counting Networks. *Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, pp. 417–428, May 1992.
8. Kruskal, C., Rudolph, L., Snir, M.: Efficient Synchronization on Multiprocessors with Shared Memory. *Proceedings of the 5th Annual ACM Symposium on Principles of Distributed Computing*, pp. 218–228, Calgary, Canada, August 1986.
9. Mavronicolas, M., Merritt, M., Taubenfeld, G.: Sequentially Consistent versus Linearizable Counting Networks. *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing*, pp. 133–142, May 1999.
10. Moran, S., Taubenfeld, G.: A Lower Bound on Wait-Free Counting. *Journal of Algorithms*, **24** (1997) 1–19.
11. Shavit, N., Touitou, D.: Elimination trees and the Construction of Pools and Stacks. *Theory of Computing Systems*, **30** (1997) 645–670.
12. Wattenhofer, R., Widmayer, P.: An Inherent Bottleneck in Distributed Counting. *Journal of Parallel and Distributed Computing*, **49** (1998) 135–145.

# Author Index

- Aguilera, M.K. 108  
Alonso, G. 93  
Anderson, J.H. 1  
Arévalo, S. 93  
  
Barrière, L. 270  
Boldi, P. 33  
  
Chatzigiannakis, I. 285  
  
Delporte-Gallet, C. 108  
Dobrev, S. 166  
Douceur, J.R. 48  
DufLOT, M. 240  
  
Fatourou, P. 330  
Fauconnier, H. 108  
Fich, F.E. 224  
Flocchini, P. 166  
Fraigniaud, P. 270  
Fribourg, L. 240  
Fujiwara, H. 123  
  
Garg, V.K. 78  
Georgiou, C. 151  
  
Harris, T.L. 300  
Herlihy, M. 136, 209, 330  
Herman, T. 315  
Higham, L. 194  
Hoepman, J.-H. 180  
  
Inoue, M. 123  
  
Jiménez-Peris, R. 93  
Johnen, C. 224  
Joung, Y.-J. 16  
  
Kim, Y.-J. 1  
Kranakis, E. 270  
Krizanc, D. 270  
  
Liang, Z. 194  
  
Malkhi, D. 63  
Masuzawa, T. 123, 315  
Mittal, N. 78  
  
Nikoletseas, S. 285  
  
Patiño-Martínez, M. 93  
Pavlov, E. 63  
Peleg, D. 255  
Picaronny, C. 240  
Pincas, U. 255  
Prencipe, G. 166  
  
Rajsbaum, S. 136  
Russell, A. 151  
  
Santoro, N. 166  
Sella, Y. 63  
Shvartsman, A.A. 151  
Spirakis, P. 285  
  
Tirthapura, S. 209  
Toueg, S. 108  
Tuttle, M. 136  
  
Umetani, S. 123  
  
Vigna, S. 33  
  
Wattenhofer, R.P. 48